palmsource™

Memory, Databases, and Files

Exploring Palm OS®

Written by Greg Wilson
Edited by Jean Ostrem
Technical assistance from Arve Hjonnevag, , Raj Mojumder, Justin Morey, Jie Su

Exploring Palm OS: Memory, Databases, and Files
Document Number 3108-003
November 9, 2004
For the latest version of this document, visit
http://www.palmos.com/dev/support/docs/.

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

Table of Contents

Part II: Reference

About This Document

This book documents Palm OS® databases, how memory is managed in Palm OS, and how Palm OS applications can use the Virtual File System to access files on expansion media.

The *Exploring Palm OS* Series

This book is a part of the *Exploring Palm OS* series. Together, the books in this series document and explain how to use the APIs exposed to third-party developers by the fully ARM-native versions of Palm OS, beginning with Palm OS Cobalt. Each of the books in the *Exploring Palm OS* series explains one aspect of the Palm operating system, and contains both conceptual and reference documentation for the pertinent technology.

IMPORTANT: The *Exploring Palm OS* series is intended for developers creating native applications for Palm OS Cobalt. If you are interested in developing applications that work through PACE and that also run on earlier Palm OS releases, read the latest versions of the *Palm OS Programmer's API Reference* and *Palm OS Programmer's Companion* instead.

As of this writing, the complete *Exploring Palm OS* series consists of the following titles:

- *Exploring Palm OS: Programming Basics*
- *Exploring Palm OS: Memory, Databases, and Files*
- *Exploring Palm OS: User Interface*
- *Exploring Palm OS: User Interface Guidelines* (coming soon)
- *Exploring Palm OS: System Management*
- *Exploring Palm OS: Text and Localization*
- *Exploring Palm OS: Input Services*
- *Exploring Palm OS: High-Level Communications*
- *Exploring Palm OS: Low-Level Communications*

- *Exploring Palm OS: Telephony and SMS*
- *Exploring Palm OS: Multimedia*
- *Exploring Palm OS: Security and Cryptography*
- *Exploring Palm OS: Creating a FEP* (coming soon)
- *Exploring Palm OS: Porting Applications to Palm OS Cobalt*
- *Exploring Palm OS: Palm OS File Formats* (coming soon)

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/

Changes to This Document

This section describes the changes made in each version of this document.

3108-002

Minor editorial corrections.

3108-001

The first release of this document for Palm OS Cobalt, version 6.0.

Part I
Concepts

This part contains conceptual and "how to" information on the Palm OS® memory system; the Data Manager, file streaming, and the VFS Manager. The Data Manger manages databases used to contain both programs and data. File streaming presents another way to access the contents of large "classic" Palm OS databases. And the VFS Manager allows you to work with the contents of files on expansion media: SD cards, Memory Stick media, and the like.

The conceptual material is organized into the following chapters:

1

Memory

This chapter helps you understand memory use on Palm OS®.

- Memory Architecture discusses how memory is structured on Palm OS. It examines the structure of the basic building blocks of Palm OS memory: heaps, chunks, and records.
- The Memory Manager discusses how to use the Palm OS Memory Manager in your applications.

IMPORTANT: Do not confuse the handheld's RAM with read/write memory on expansion cards, such as SD cards or Memory Stick media. You access expansion cards through a different API. See Chapter 3, "Virtual File Systems," on page 69 for more information.

Memory Architecture

IMPORTANT: This section describes the current implementation of Palm OS memory architecture. This implementation may change as Palm OS evolves. Do not rely on implementation-specific information described here; instead, always use the API provided to manipulate memory.

The Palm OS divides the total available RAM store into two logical areas: **dynamic heaps** and the **storage heaps**. A process's dynamic heap is used as working space for temporary allocations, and is analogous to the RAM installed in a typical desktop system. RAM not reserved for dynamic use is designated for the storage heaps and is analogous to disk storage on a typical desktop system.

Because power is always applied to the memory system, the dynamic and storage heaps preserve their contents when the handheld is turned “off” (that is, when it is in low-power sleep mode). Storage heaps are preserved even when the handheld is explicitly reset (unless the user performs a hard reset, in which case the storage heaps are reinitialized).

The Dynamic Heaps

The dynamic heap provides memory for dynamic allocations. From this heap the system provides memory for dynamic data such as global variables, system dynamic allocations, application stacks, temporary memory allocations, and application dynamic allocations (such as those performed when the application calls `malloc()` or `MemHandleNew()`). Each process has an independent dynamic heap that is created and destroyed along with the process.

The entire amount of RAM reserved for the dynamic heaps is always dedicated to this use, regardless of whether it is actually used for allocations. The size of the dynamic area of RAM on a particular handheld varies according to the OS version running, the amount of physical RAM available, the requirements of pre-installed software such as the TCP/IP stack or IrDA stack, and any other licensee requirements.

The Storage Heaps

The remaining portion of RAM not dedicated to use by the dynamic heaps is configured as a set of storage heaps and is used to hold nonvolatile user data such as appointments, to do lists, memos, address lists, and so on. An application accesses a storage heap by calling Data Manager functions such as `DmNewHandle()`. Storage heaps retain their contents through soft reset cycles.

The size of the storage heap available on a particular handheld varies according to the OS version that is running; the amount of physical RAM and ROM that is available; and the storage requirements of end-user application software such as the Address Book, Date Book, or third-party applications.

Note that you typically work with the storage heap by manipulating databases. See Chapter 2, “Palm OS Databases,” for information on creating and accessing Palm OS databases.

Heap Details

A **heap** is a contiguous area of memory used to contain and manage one or more smaller chunks of memory. When applications work with memory (for instance, allocate, resize, or free) they usually work with chunks of memory. An application can specify whether to allocate a new chunk of memory in a dynamic heap or a storage heap. The Memory Manager and the Data Manager each manages their respective heaps, rearranging chunks as necessary to defragment the heaps and merge free space.

Heaps in the Palm OS environment are referenced through heap IDs. A **heap ID** is a unique 16-bit value that is used to identify a heap within the Palm OS address space. The three defined heaps IDs are:

Heap ID	Heap	Managed By
0	Dynamic heap	Memory Manager
1	Storage heap for classic and extended record databases, and extended resource databases except for those that contain ARM-native code.	Data Manager
2	ROM heap	Data Manager
3	Storage heap for schema databases and resource databases containing ARM-native code.	Data Manager

Chunks

In the Palm OS environment, all data are stored in chunks. A **chunk** is an area of contiguous memory between 1 byte and slightly less 2^26 bytes in a storage heap, or 2^31 bytes in a dynamic heap.

Every memory chunk used to hold storage data (as opposed to memory chunks that store dynamic data) is a record in a database implemented by the Palm OS Data Manager.

Memory chunks can be movable or immovable. When working with a storage heap, applications need to store data in movable chunks whenever feasible, thereby allowing the operating system to move chunks as necessary to create contiguous free space in memory for allocation requests. In a dynamic heap, on the other hand, applications should use the standard C APIs for working with memory (`malloc()`, `free()`, and the like); the standard C APIs have no concept of movable chunks.

When an application requests an immovable chunk it receives a pointer to that chunk. The pointer is simply that chunk's address in memory. Because the chunk cannot move, its pointer remains valid for the chunk's lifetime; thus, the pointer can be passed "as is" to the caller that requested the allocation.

When an application requests a movable chunk, the operating system generates a pointer to that chunk, just as it did for the immovable chunk, but it does not return the pointer to the caller. Instead, it stores the pointer to the chunk, called the **master chunk pointer**, in a **master pointer table** that is used to track all of the movable chunks in the heap, and returns a reference to the master chunk pointer. This reference to the master chunk pointer is known as a **handle**. It is this handle that the operating system returns to the caller that requested the allocation of a movable chunk.

Using handles imposes a slight performance penalty over direct pointer access but permits the operating system to move chunks around in the heap without invalidating any chunk references that an application might have stored away. As long as an application uses handles to reference data, only the master pointer to a chunk needs to be updated when the chunk is moved during heap defragmentation.

An application typically **locks** a chunk handle for a short time while it has to read or manipulate the contents of the chunk. The process of locking a chunk tells the Memory or Data Manager to mark that data chunk as immobile; a pointer to the chunk is returned that your application can use to manipulate the chunk contents. When an application no longer needs the data chunk, it should unlock the handle immediately to keep heap fragmentation to a minimum.

Chunks maintain a lock count. A count of zero indicates that the chunk is movable. Every time you lock a chunk, its lock count is incremented. You can lock a chunk a maximum of 14 times before an error is returned. (Unmovable chunks hold the value 15 in the lock field.) Unlocking a chunk decrements the value of the lock field by 1. When the lock count is reduced to 0, the chunk is again free to be moved by the operating system.

IMPORTANT: Note that any handle is good only until the system is reset. When the system resets, it reinitializes all dynamic memory areas and relaunches applications. Therefore, *you must not store a handle in a database*.

Internally each chunk is located by means of a **local ID**. The local ID of immovable chunk is a pointer to the chunk; the local ID of movable chunk is equivalent to the chunk handle.

Owner ID

In previous versions of Palm OS, the operating system used an **owner ID** to associate that chunk with an application. Because the dynamic heap of the main UI application is always destroyed and recreated during an application switch, owner IDs of memory chunks don't make sense in Palm OS Cobalt. The Memory Manager APIs for managing owner IDs exist for compatibility reasons; setting the owner ID of a chunk doesn't make the chunk persistent across application switches as in previous versions of Palm OS.

The Memory Manager

The Palm OS Memory Manager is responsible for maintaining the location and size of every memory chunk in the dynamic heaps. It provides an API for allocating new chunks, disposing of chunks, resizing chunks, locking and unlocking chunks, and compacting heaps when they become fragmented.

IMPORTANT: In Palm OS Cobalt the Memory Manager APIs exist mainly for use by the Data Manager to manage storage heaps. Application developers should use the standard C library functions such as `malloc()` and `free()` to manage dynamic memory.

Allocating and Freeing Memory Chunks

To allocate a movable chunk, call `MemHandleNew()` and pass the desired chunk size. To free a memory chunk given its handle, call `MemHandleFree()`. The Memory Manager provides similar functions that work with immovable chunks: `MemPtrNew()` allocates a memory chunk and returns a pointer to it, while `MemPtrFree()` frees a chunk given its pointer.

NOTE: You cannot allocate a zero-size chunk.

The size of a chunk can be obtained with either `MemHandleSize()` or `MemPtrSize()`, depending on whether you have the chunk's handle or a pointer to its data. To resize a movable chunk use `MemHandleResize()`. When resizing immovable chunks `MemPtrRealloc()` is recommended; although there is a function called `MemPtrResize()`, it should only be relied upon when you are making the chunk smaller since it can't increase the size of an immovable chunk unless there is free space in the heap immediately following the chunk.

If you have a pointer to a locked, movable chunk, you can recover the handle by calling `MemPtrRecoverHandle()`.

Manipulating Chunk Contents

Because you have a pointer to any immovable chunk you've allocated with `MemPtrNew`, you can read or write that chunk's contents directly. Before you can read or write data to a movable chunk, however, you must call `MemHandleLock()` to lock it and get a pointer to it. Then, when you no longer need direct access to the chunk's contents, call `MemHandleUnlock()`. (Note that after a call to `MemHandleUnlock`, the pointer your application was using to access the chunk's contents is no longer valid.)

The Memory Manager provides three utility functions that you can use when working with the contents of a chunk:

- `MemMove()` moves memory from one place to another.
- `MemSet()` fills memory with a specific value.
- `MemCmp()` compares two regions of memory.

Note that in Palm OS Cobalt, however, applications should normally use the standard C library functions such as `memmove()` or `memcpy()`, `memset()`, and `memcmp()` to manage dynamic memory.

Summary of Memory Management

Memory Manager Functions

Allocating and Freeing Memory

MemHandleFree()	MemPtrFree()
MemHandleLock()	MemPtrNew()
MemHandleNew()	MemPtrUnlock()
MemHandleUnlock()	

Resizing Chunks

MemHandleResize()	MemPtrResize()
MemHandleSize()	MemPtrSize()
MemPtrRealloc()	

Working With Memory

MemCmp()	MemDynHeapReleaseUnused()
MemMove()	MemHeapCompact()
MemSet()	

Chunk Information

MemHandleDataStorage()	MemPtrDataStorage()
MemHandleHeapID()	MemPtrRecoverHandle()
MemHandleSetOwner()	MemPtrSetOwner()

Heap Information

MemDynHeapGetInfo()	MemHeapID()
MemDynHeapOption()	MemHeapSize()
MemHeapCheck()	MemNumHeaps()
MemHeapDynamic()	MemNumRAMHeaps()
MemHeapFlags()	MemPtrHeapID()
MemHeapFreeBytes()	

Memory Manager Functions	
Debugging	
`MemDebugMode()`	`MemHeapScramble()`
`MemSetDebugMode()`	

2

Palm OS Databases

This chapter describes how to work with Palm OS® databases. Two separate header files declare the APIs you use: `SchemaDatabases.h` (documented in Chapter 7, "Schema Databases," on page 291) and `DataMgr.h` (documented in Chapter 4, "Data Manager," on page 99). In addition, the File Streaming APIs, which allow you to access classic databases using a mechanism very similar to UNIX file streams, are declared in `FileStream.h` (and documented in Chapter 5, "File Stream," on page 239).

This chapter is divided into the following major sections:

IMPORTANT: To access data or resources on secondary storage (such as expansion cards), you use a different set of APIs. See Chapter 3, "Virtual File Systems," on page 69 for more information.

Database Overview

A traditional file system first reads all or a portion of a file into a memory buffer from disk, using or updating the information in the memory buffer, and then writes the updated memory buffer back to disk. Because Palm Powered™ handhelds have limited amounts of dynamic RAM and use nonvolatile RAM instead of disk storage, a traditional file system is not optimal for storing and retrieving Palm OS user data. Thus, except when working with expansion media (an SD card, Memory Stick, and the like), Palm OS doesn't make use of a traditional file system. Instead of files, Palm OS applications work with **databases**.

Databases organize related rows (for schema databases) or records (for non-schema databases); each belongs to one and only one database. A database may be a collection of all address book entries, all datebook entries, and so on. A Palm OS application can create, delete, open, and close databases as necessary, just as a traditional file system can create, delete, open, and close a traditional file.

For those new to Palm OS programming, the term “database” can be somewhat misleading. Palm OS Cobalt supports three different types of database, some of which look more like conventional databases than others. Schema databases, which were introduced in Palm OS Cobalt, bear a strong resemblance to relational databases. Data is organized into tables, which consist of rows and columns. **Schema databases** use the concept of a **schema** to define the structure of a table row. Unlike relational databases, however, schema databases don’t allow you to perform joins and other complex operations.

The other two database types are classified as “non-schema” databases because they are significantly less structured. There are two supported non-schema database types:

- **Classic databases** are supported for compatibility with earlier versions of Palm OS. All versions of Palm OS back to Palm OS 1.0 support this database format, and this is the format used by applications running on Palm OS Cobalt through PACE.
- **Extended databases** are an “extended” version of classic databases. There are three primary differences between classic and extended databases: extended databases records can exceed 64K in length (classic records cannot); extended databases are uniquely identified by a combination of name and creator ID (classic databases are uniquely identified by name alone); and extended databases can store data using the processor’s native endianness (classic databases must store record data using big-endianness, for compatibility with the 68K-based Dragonball CPU used in the early Palm OS devices).

Palm OS Cobalt applications that must remain compatible with an earlier OS release—perhaps a version of the application exists that runs on earlier versions of Palm OS and this application must be able to work with the earlier version's data—will use classic databases. Those Palm OS Cobalt applications that don't have such a compatibility requirement should use either extended or schema databases instead. Which to use depends on the nature of the application. Schema databases provide a great deal of support for organizing the database contents and for security, at the expense of performance. Extended databases, on the other hand, are faster to read and write, but less secure and less structured—meaning that your application has to do the work of maintaining and interpreting record contents itself.

Schema Databases

Non-schema databases treat their contents as lists of mostly opaque records. The Data Manager knows just enough about each record to understand category assignment, modification status, and deletion status. Applications are entirely responsible for structuring and interpreting database record contents. Traditional Palm OS applications, written for 68K-based handhelds and for PACE, work exclusively with classic databases.

Schema databases add a layer of abstraction to the record contents. This extra layer of abstraction allows you to create more flexible applications, with improved sharing of data between applications. Because the Data Manager knows more about the structure of the database rows, it can provide additional capabilities, such as system-managed, optimized, and internationalized sorting. It lets you bind variables to various row fields, so as you move from one row to another the bound variables are automatically updated with the contents of the corresponding row's fields. And, you can create **cursors**, subsets of a database table's rows selected and sorted based upon application-specific criteria. Schema databases have other advantages as well:

- They provide more standardized data storage.
- Schema databases make synchronization simpler and more efficient.
- Schema databases can be more easily extended with additional fields.

- It is much easier to create conduits for schema databases, and it is easier to integrate a schema database with a database on the desktop computer or on a server.

Resources and Resource Databases

Applications can use the Data Manager to retrieve and save chunks of data conveniently. Non-schema databases that are designated as resource databases tag each chunk of data with a unique resource type and resource ID. These tagged data chunks are called **resources**. Resource databases are almost identical in structure to other non-schema databases except for a slight amount of increased storage overhead per resource record (two extra bytes).

Resources are typically used to store the user interface elements of an application, such as images, fonts, dialog layouts, and so forth. Part of building an application involves creating these resources and merging them with the actual executable code. In the Palm OS environment, an application is, in fact, simply a resource database with the executable code stored as one or more code resources and the graphics elements and other miscellaneous data stored in the same database as other resource types.

Applications may also find resource databases useful for storing and retrieving application preferences, saved window positions, state information, and so forth. These preferences settings can be stored in a separate resource database.

Uniquely Identifying Databases

As in previous releases of Palm OS, classic databases must be uniquely identified by name. Schema and extended databases, however, are uniquely identified by a combination of the database's name and its creator ID. Thus, schema and extended database names need only be unique for a single creator ID: two such databases with the same name can reside on a single handheld as long as their creator IDs differ.

Database Attributes

In addition to the records that make up the database's contents—and in addition to the schemas that define the structure of the rows in a schema database table—all Palm OS databases have a set of flags that describe various aspects of the database itself, plus a set of dates identifying when the database was created, last modified, and last backed up. As well, non-schema databases have an **Application Info block** to hold application settings and the like, and a **Sort Info block** to control the ordering of database records (schema databases use a different mechanism to control row ordering; see "Cursors" on page 36).

You obtain the database attribute flags and dates, along with handles for the Sort Info block and the Application Info block if working with a non-schema database, by calling `DmDatabaseInfo()`.

Automatic Database Backup and Restore

Palm OS Cobalt version 6.1 can be configured by a licensee to back up the contents of the RAM storage heaps to some sort of non-volatile NAND flash. In the event that the RAM storage heaps are corrupted or are lost for some reason, the storage heaps can then be restored to their saved state. This provides an additional level of data reliability beyond what's already provided by HotSync. Devices without backup batteries may take advantage of this backup and restore capability to prevent data loss between power on/off sessions.

For security, the backup is performed to a private internal VFS volume that can only be accessed by the Data Manager, only for purposes of backup and restore.

Backup is triggered on a limited set of events:

- Database close. Any time that a database is closed, the database is backed up to the non-volatile store.
- Database create. Upon creation, the database is backed up. This takes care of installed databases that are never modified and thus not otherwise backed up.
- A successful call to `DmSetDatabaseInfo()`. Whenever a call to `DmSetDatabaseInfo()` succeeds, the database information is backed up to the non-volatile store.
- Device sleep. Whenever the device goes to sleep as a result of the normal system sleep functionality, the Data Manager

iterates through all open databases and backs them up to the non-volatile store. This takes care of those databases that are opened by an application and not closed until the application exits, and those databases that are opened by background threads that are running when the system goes to sleep.

- An explicit call to `DmInitiateAutoBackupOfOpenDatabase()`.

Every time the device resets with an indication that the contents of RAM may have been lost, the backup volume is restored to RAM. Before restoring the backup contents, a consistency check is performed on the backup and an attempt is made to fix any inconsistencies. Databases are only restored under these circumstances; developers cannot trigger a database restore programmatically.

Working with Schema Databases

Schema databases consist of one or more tables. All of the rows in a given table have the same structure.

All data in a schema database table is represented in the form of two-dimensional tables. A table contains zero or more rows and one or more columns. All rows in a table have the same sequence of columns, but with a different series of values in those columns. Note that a row doesn't have to have a value for a column; the special value `NULL` can be used to indicate that the value is undefined.

As with a relational database, operations are defined by logic, not by the position of a row within a table. That is, you ask for all rows where (x = 3) and not for the first, third, and fifth rows, for example. The rows of a schema database table are in arbitrary order—the order in which they appear doesn't necessarily reflect the order in which they were entered or in which they are stored.

One of the strengths of the relational approach (which applies to schema databases) is that you can deal with the data as information and, ideally, not worry about the details of how it is represented or physically maintained in the database itself. Having to deal with these kinds of implementation details makes extended and classic databases more difficult to manage.

Schemas and Tables

In Palm OS Cobalt, a **schema** is simply the collective definitions of a table's columns. While there is no single structure or identifier that represents a schema, the `DbTableDefinitionType` structure contains a count of the number of columns in the table and a series of pointers to the structures that define those table columns: essentially, the schema (this structure also contains the table's name, which isn't part of the schema itself).

Each schema database can be heterogeneous in that it can support multiple tables. Because each table's definition includes the column definitions for that table—the schema—two tables can have the same schema, yet changes to one table's schema doesn't affect the other.

Tables can be defined at the time a database is created, or added later.

Schema access is gated by the access restrictions for the database. Read-only access to a database implies read-only access to all of that database's schemas (and thus any attempt to modify the schema will fail). See "Secure Databases" on page 47 for more information on database access restrictions.

Logical (External) vs. Physical (Internal) Views

Schemas allow the Data Manager to decouple the logical (external) view of your data from the physical (internal) view. When working with a schema database you manipulate row data in terms of data types defined in the column property sets—this is the **logical data view**. In actual fact, however, the Data Manager stores row data internally in an unpublished variant format: the **physical data view**. This decoupling facilitates changes to internal data formats without affecting existing database consumers.

Data types defined in column property sets are Palm OS primitives or their vectors. The Data Manager converts between its physical data types and the logical data types that are enforced during field get and set operations.

Column Properties

A schema is a collection of column property sets. A column property set is represented as a `DbSchemaColumnDefnType` structure. This structure contains the following:

ID: A 32-bit application-defined identifier. This ID must be unique for a given table.

Name: An application-defined name for the column. The column name must be unique for a given table. It can be up to 32 bytes in length, including the terminating null character, and must be a valid SQL identifier consisting only of 7-bit ASCII characters. The column name is stored in a single application-defined language encoding.

Data Type: The type of data contained within the database column.

Size: The size, in bytes, for the column. For columns that contain variable-length strings, blobs, and vectors, this is the maximum size of the string, blob, or vector. For all other types this is the actual size of the type.

Attributes: A set of flags that indicate whether the column data can be modified, whether the column was added to the table after the table was created, and whether or not the column data will be synchronized. (Modifications made to a "non-syncable" column's data don't change the modification state for the row, and thus by themselves don't cause the row to be synchronized during a HotSync operation.)

These are built-in column properties provided by the Data Manager. In addition to these built-in properties, you can define custom properties for a column: properties that facilitate application-specific semantics for columns. For more information on manipulating the column definitions that make up a schema, see "Working with Column Definitions" on page 23.

Column Data Types

Palm OS Cobalt schema databases support the column data types listed in Table 2.1.

Table 2.1 Supported schema column data types

Palm Primitive/ Logical Types	Description	Storage Requirement	Range/Size
`uint8_t`	Unsigned char	1 byte	0 to 255
`uint16_t`	Unsigned short int	2 bytes	0 to 65535
`uint32_t`	Unsigned int	4 bytes	0 to 4294967295
`uint64_t`		8 bytes	
`int8_t`	Signed char	1 byte	-128 to 127
`int16_t`	Signed short int	2 bytes	-32768 to 32767
`int32_t`	Signed int	4 bytes	-2147483648 to 2147483647
`int64_t`		8 bytes	
`float`	Float	4 bytes	
`double`	Double	8 bytes	
`Boolean`	True /False value	1 byte	0 or 1
`DateTimeType`	Date-Time type	14 bytes	
`DateType`	Date expressed as an absolute date	2 bytes	
`TimeType`		2 bytes	
`time_t`	(`dbDateTimeSecs`) Time in seconds since the UNIX epoch	4 bytes	-2147483648 to 2147483647
`char`	Fixed-length character string	*m* bytes, where m is the statically-defined length and 1 <= *m* <= 255	1 <= *m* <= 255, where *m* is the maximum defined length.

Table 2.1 Supported schema column data types *(continued)*

Palm Primitive/ Logical Types	Description	Storage Requirement	Range/Size
VarChar	Variable-length character string	*n*+4, where *n* is the actual string length and where *n* <= *m*. *m* is the maximum defined length and 1 <= *m* <= 2^{32}	1 <= *m* <= 2^{32}, where *m* is the maximum defined length.
blob	Variable-length array of bytes.	*n*+4, where *n* is the actual string length and where *n* <= *m*. *m* is the maximum defined length and 1 <= *m* <= 2^{32}	1 <= *m* <= 2^{32}, where *m* is the maximum defined length.
Vector	Variable-length vectors of Palm primitive numeric, string, and date-time types. See Table 2.2, below, for a list of supported vector types.	*n*+4, where *n* is the number of bytes needed to contain the vector.	2^{32} bytes.

Table 2.2 Supported vector types

Vector Types	Usage
`uint8_t` vectors	`uint8_t[]`
`uint16_t` vectors	`uint16_t[]`
`uint32_t` vectors	`uint32_t[]`

Table 2.2 Supported vector types *(continued)*

Vector Types	Usage
`uint64_t` vectors	`uint64_t[]`
`float` vectors	`float[]`
`double` vectors	`double[]`
`Boolean` vectors	`Boolean[]`
`DateTimeType` vectors	`DateTimeType[]`
`DateType` vectors	`DateType[]`
`TimeType` vectors	`TimeType[]`
String vectors	Array of null-terminated strings, with an extra terminating null character marking the end of the vector. For instance, using 7-bit ASCII: `"String1\0String2\0String3\0\0"`

NOTE: In a string vector, the null characters must be interpreted as encoding-dependent null characters instead of null bytes. A null character may be multi-byte for a specific encoding scheme.

Database, Table, and Column Identifiers

Schema databases are uniquely identified by a combination of their name and their creator code. However, most of the schema database functions take database identifiers of the type `DatabaseID`. The function `DmFindDatabase()` returns a database ID for an existing database, while `DbCreateDatabase()` creates a new database (given a name, creator code, and type) and returns a database ID for the newly-created database.

Database tables are identified by name. There is no need for a numeric "table identifier." However, each database does maintain an array of tables that you can access by index. This array is zero-based; its indices range from zero to *n*-1, where *n* is the number of tables defined for that database. This value can be obtained by calling `DbNumTables()`. Given the index of a table within a database, you can translate it into the table's name by calling `DbGetTableName()`.

A column is uniquely identified by either the column's descriptive name or by a 32-bit ID (both must be unique). These application-defined column names and IDs allow multiple applications within a given application context to share a common semantic understanding of a given column type. For instance, two applications might select a name of "EMNO" for the employee number column of the "EMPLOYEE" database and use column-based search and retrieval of values in the column named "EMNO". The design-time specification of both column identifiers and table names facilitates the development of public metadata interfaces for databases and encourages generic data exchange based on these interfaces.

As with tables in a database, you can iterate through the columns in a table. To obtain the number of columns in a given table, call `DbNumColumns()`. You can retrieve the definitions for each of the columns in the row by calling `DbGetColumnDefinitions()`. To obtain the ID of an individual column given its index (which again ranges from 0 to *n*-1, where *n* is the number of columns in the table), use `DbGetColumnID()`.

Creating, Modifying, and Deleting Tables

You can create tables either at the time you create a database or after the fact. Each table is a `DbTableDefinitionType` structure; this structure contains the table's name and an array of column definitions. Allocate memory as needed for the `DbTableDefinitionType` structures (and for the `DbSchemaColumnDefnType` structures needed to define the table's columns), and initialize them as appropriate for your application. Then, either supply them when creating your database (with `DbCreateDatabase()` or `DbCreateSecureDatabase()`, as appropriate), or add them to an existing database with `DbAddTable()`.

You can remove a table from a database only if the table contains no non-deleted rows. If the table contains non-deleted rows, create a cursor that selects all of the table's rows, and then call

`DbCursorRemoveAllRows()`. Once the table is empty, call `DbRemoveTable()` to remove the table from the database.

When modifying an existing table, you are limited to adding and removing columns and modifying custom column properties. Get the existing table definition by calling `DbGetTableSchema()`. Use `DbAddColumn()` to add a a column to an existing table.

Working with Column Definitions

Each table maintains a list of column definitions. As discussed in "Database, Table, and Column Identifiers" on page 21, given an index into that list you can obtain the corresponding column ID. This ID is necessary to work with individual columns, but isn't needed to obtain the complete set of column definitions that make up a schema.

To obtain the column definitions for a table, you can use one of two functions. `DbGetAllColumnDefinitions()` retrieves all column definitions for the specified table, while `DbGetColumnDefinitions()` retrieves one or more column definitions for the table—supply an array of column IDs indicating which column definitions are to be retrieved. Both functions return an array of column definitions (`DbSchemaColumnDefnType` structures); when you are done with this array you must release the memory consumed by the array with a call to `DbReleaseStorage()`.

In addition to any custom properties you define for a column definition, all columns have a set of built-in properties. These built-in properties are read-only, to prevent applications from modifying existing data row columns in a way that can impact other data consumers. Each built-in property has a corresponding constant definition that can be used as input to a generic accessor—`DbGetColumnPropertyValue()`—that retrieves the value of the specified column property. The constant definitions for the built-in properties are predefined; see "DbSchemaColumnProperty" on page 295 for the constants themselves. The following are the built-in properties for a column:

- Name (must be unique)
- Data type

- Size (maximum byte size for variable-length strings, blobs, and vectors)
- Attributes

Unlike the built-in properties, custom properties may be read, written and deleted. Custom property IDs must fall outside the built-in property ID range. That is, they must be greater than `dbColumnPropertyUpperBound`.

For a given column, define custom properties using `DbSetColumnPropertyValue()` or `DbSetColumnPropertyValues()`. If the specified property ID does not exist, a custom property is created with the specified ID and value. If the specified property ID exists, its value is updated to the new value.

The value of any property—whether built-in or custom—can be obtained by calling either `DbGetColumnPropertyValue()`, to obtain a single property value, or `DbGetColumnPropertyValues()` to obtain multiple property values at one time.

To remove a property from a given column, call `DbRemoveColumnProperty()`. Note that this function is very different from `DbRemoveColumn()`: whereas `DbRemoveColumnProperty()` removes only a property from a column, `DbRemoveColumn()` removes an entire column from a table, *along with that column's data*.

Row Attributes

Schema database rows can have the attributes listed in Table 2.3.

Table 2.3 Schema database row attributes

Attribute	Description
`dbRecAttrArchive`	The row's data is preserved until the next HotSync. When the `dbRecAttrArchive` bit is set, the `dbRecAttrDelete` bit is set as well, so archived rows are otherwise treated like deleted rows.
`dbRecAttrDelete`	The row has been deleted.
`dbRecAttrReadOnly`	The row is read-only, and cannot be written to.
`dbRecAttrSecret`	The row is private.

NOTE: The Data Manager does not place any semantics on the read-only attribute. It is up to the application to enforce the read-only semantics.

The read-only attribute is used to support certain record sharing scenarios that allow a user to view a record, but not to modify it. Note that schemas also allow the definition of "always writable" columns that allow particular fields to be writable in a read-only row. This might be used, for example, in a calendar event for a TV show that is read-only (you can't reschedule the show); the field containing the alarm information would be "always writable" allowing each user the option of setting an alarm.

Table 2.4 lists the functions that you use to get and set a schema database row's row ID, category, and attributes.

Table 2.4 Functions used to access row information

Category	Functions

Local ID	`DbCursorGetCurrentRowID()`
	`DbCursorGetRowIDForPosition()`

Category Membership	`DbAddCategory()`
	`DbGetCategory()`
	`DbIsRowInCategory()`
	`DbNumCategory()`
	`DbRemoveCategory()`
	`DbSetCategory()`

Attributes	`DbGetRowAttr()`
	`DbSetRowAttr()`

Categories

Categories are a user-controlled means of grouping or filtering records or rows. Non-schema databases allow records to be a member of only one of 15 categories, or "Unfiled." Schema database rows, on the other hand, can be a member of any combination of up to 255 categories (or none—the equivalent of "Unfiled"). Thus, where in a extended database a record might, say, have to either fall into the "Personal" or "Business" category, in a schema database a row could fall into both.

As with non-schema databases, category information is local to a database. However, unlike non-schema databases which store information about that database's categories in the Application Info block, schema databases rely upon an internal "category info" block to contain this information.

Information about the database's categories, such as the number and names of the categories, as well as the order in which they occur in a UI list, is controlled by the Category Manager. The Data Manager is only responsible for managing the category membership of individual database rows.

Category membership for a row is limited to the maximum number of categories that can be defined locally in a schema database. Since the maximum number of categories a database can support is limited to 255, any given row can only be a member of up to 255 categories.

In a non-schema database, records are always in one category ("Unfiled" is just a specific category). In a schema database, rows may be in one category, multiple categories, or none. The notion of "Unfiled" as a category doesn't make sense here since rows shouldn't be able to be in the "Unfiled" category and in other categories at the same time. Because applications can display or perform other operations on rows with no category membership, a row that is a member of no database categories could be thought of as "Unfiled." Note that the Category Manager controls how rows with no category membership are displayed to end users.

The Data Manager stores category IDs as category membership information for a record or row. Storing category IDs abstracts the Data Manager from any modifications performed on the internal category structure, such as adding or deleting a category.

The following functions let you manipulate a schema database row's category membership:

`DbSetCategory()`
: Sets category membership for a single database row.

`DbAddCategory()`
: Makes the specified row a member of one or more additional categories.

`DbGetCategory()`
: Retrieves the category membership for the specified row.

`DbNumCategory()`
: For a specified row, determines how many categories the row is a member of.

`DbRemoveCategory()`
: Removes category membership in the specified categories from a single row.

These functions let you manipulate rows that meet the given category membership criteria:

`DbIsRowInCategory()`
: Determines if a row has membership for the specified categories, depending on the given match mode criteria.

`DbMoveCategory()`
: Replaces one or more categories with the specified category for all rows, depending on the given match mode criteria.

`DbRemoveCategoryAllRows()`
: Removes category membership in the specified categories from all rows in the database, depending on the match mode criteria.

`DbCursorOpenWithCategory()`
: Creates and opens a cursor containing all rows in the specified table that conform to a specified set of flags, ordered as specified. Rows are filtered based upon category membership.

The Application Info Block

Schema databases don't have a dedicated Application Info block. For application-specific data of the type found in a non-schema database's Application Info block, create a database table specifically for this purpose.

Schema Database Rows

As discussed in "Schemas and Tables" on page 17, a schema database table can have zero or more rows, and each row within the table shares a common structure, or schema.

Rows are identified by a 32-bit identifier that is unique within the database. You supply the row ID (or, often, the cursor ID as discussed under "Cursors" on page 36) when archiving rows, copying row contents, deleting rows, and the like. In the rare instance that you find yourself with a row ID independent of the table from which it came, you can determine to which table the row belongs by calling `DbGetTableForRow()`.

Creating New Rows

To create a row, construct an array of `DbSchemaColumnValueType` structures, one for each of the row's values. To add your row to a table (you can't add a row to a database without adding it to a database table), you pass the structures to `DbInsertRow()`. Assuming that the row was added to the table successfully, this function returns the row ID of your new row. Optionally, you can add an “empty” row by calling `DbInsertRow()` without supplying the `DbSchemaColumnValueType` structures. See the description of `DbInsertRow()` for more information.

Rows added to a table are added to the end of the database. You aren’t given the opportunity to specify the position of the row within the table. The schema database APIs also don’t include a function for altering the position of a row within a table. That is because when working with schema database rows you often are working within the context of a **cursor**, within which you can perform such operations.

Reading Data

Columns in a row are identified either by a 32-bit application-defined ID or by an index. The index is zero-based and ranges from 0 <= *index* < *n*, where *n* is the number of columns in the schema. Note that the index of columns added after the schema is initially created may change, so do not make persistent references to table columns by their index.

Individual row column values may only be extracted using column IDs. The Data Manager provides a function that returns a column’s ID given its index: `DbGetColumnID()`.

`DbGetColumnValue()` retrieves a single column value. This function is restrictive, however, in the sense that it does not allow value retrieval into user-allocated buffers but always returns a reference to a storage heap buffer. Also, for greater efficiency most applications will want to retrieve multiple columns using either `DbGetColumnValues()` or `DbCopyColumnValues()`.

For columns containing string or vector data, you can retrieve partial column values through the use of an offset. This is useful for columns containing large strings or blobs where, for space efficiency it makes sense to only read or write a portion of the column’s data.

When retrieving values, you can retrieve them either by copy or by reference.

Value Copy: You allocate output buffers, enclose each in a `DbSchemaColumnValueType` structure, and pass them to the Data Manager by calling either `DbCopyColumnValue()`

or DbCopyColumnValues(). The Data Manager then copies column data into the buffers.

Value Reference: You call either DbGetAllColumnValues(), DbGetColumnValues(), or DbGetColumnValue(), and receive back references to column data. This saves RAM by not requiring an additional buffer for column value storage. When you are done working with the data, you must explicitly release the Data Manager-allocated buffer with DbReleaseStorage(), which unlocks the row.

The storage locality of the buffers for the various value retrieval functions is detailed in Table 2.5 for different database types.

Table 2.5 Buffer storage locality for column value retrieval functions

Function	Non-Secure	Secure
`DbGetAllColumnValues()` `DbGetColumnValues()` `DbGetColumnValue()`	Data Manager returns references to storage-heap-based column values.	Data Manager returns references to dynamic-heap-based column values. References to storage heap values are not returned for secure databases.
`DbCopyColumnValue()` `DbCopyColumnValues()`	Data Manager copies column values to user-allocated dynamic heap storage.	Data Manager copies column values into user-allocated dynamic heap storage.

The code excerpt in Listing 2.1 illustrates how you can retrieve a single column value with `DbGetColumnValue()`.

Listing 2.1 Retrieving a single column value

```
status_t errCode;
char nameP[25];
void *valueP;
uint32_t valueSize;
uint32_t columnID = 768;
```

```
errCode = DbGetColumnValue(dbRef, rowID, columnID, 0,
   &valueP, &valueSize);
if (errNone == errCode){
   // process each column value
   memcpy(nameP, valueP, valueSize);
} else {
   ErrDisplay("Error in retrieving column value");
   return errCode;
}

// release storage heap buffer returned by the Data Manager
DbReleaseStorage(dbRef, valueP);
```

The code in Listing 2.2 is similar to the above, but it shows how to use `DbGetAllColumnValues()` to retrieve every column value for a database row with a single call.

Listing 2.2 Retrieving all column values

```
DbSchemaColumnValueType *columnValueArray;
status_t errCode;
uint32_t numColumns;

errCode = DbGetAllColumnValues(dbRef, rowID,
   &numColumns, &columnValueArray);
if (errNone == errCode){
   // iterate through the column value array
   for (int i=0; i<numColumns; i++){
      if (errNone == columnValueArray[i].errCode){
         // process each column value
      } else {
         // handle error in retrieving column value.
         ErrDisplay("Error in retrieving column value");
         break;
      }
   }
} else {
   ErrDisplay("Error in retrieving column values");
   return errCode;
}

// Release storage heap buffer returned by the Data Manager
// This invalidates all columnValueArray[i].columnData
// references.
```

```
DbReleaseStorage(dbRef, columnValueArray);
}
```

In addition to retrieving a single column value or all column values, you can set up an array of column IDs and use `DbGetColumnValues()` to retrieve a subset of the row's values. Listing 2.3 illustrates the use of `DbGetColumnValues()` in this way.

Listing 2.3 Retrieving multiple, specific column values

```
DbSchemaColumnValueType *columnValueArray;
status_t errCode;
uint32_t columnIDArray[] = {768, 770, 771};
uint32_t numColumns = sizeof(columnIDArray)/sizeof(uint32_t);

errCode = DbGetColumnValues(dbRef, rowID, numColumns,
   columnIDArray, &columnValueArray);
if (errNone == errCode){
   // iterate through the column value array
   for (int i=0; i<numColumns; i++){
      if (errNone == columnValueArray[i].errCode){
         // process each column value
      } else {
         // handle error in retrieving column value.
         ErrDisplay("Error in retrieving column");
         break;
      }
   }
} else {
   ErrDisplay("Error in retrieving column values");
   return errCode;

// Release storage heap buffer returned by the Data Manager.
// This invalidates all columnValueArray[i].columnData
// references.
DbReleaseStorage(dbRef, columnValueArray);
```

As a final example, Listing 2.4 shows how to retrieve multiple column values but have them copied into pre-allocated buffers by `DbCopyColumnValues()`.

Listing 2.4 Copying multiple, specific column values

```
DbSchemaColumnValueType columnValueArray[4];
uint32_t numColumns = sizeof(columnValueArray) /
   sizeof(DbColumnValueType);
uint32_t rowIndex;
status_t errCode;

typedef struct {
   char userName[20];
   char userAddressLine1[25];
   char userAddressLine2[25];
   char userAddressLine3[25];
} userDetailsType;
userDetailsType user;

columnValueArray[0].columnID = 768;
columnValueArray[0].data = user.userName;
columnValueArray[0].dataSize = sizeof(user.userName);

columnValueArray[1].columnID = 770;
columnValueArray[1].data = user.userAddressLine1;
columnValueArray[1].dataSize = sizeof(user.userAddressLine1);

columnValueArray[2].columnID = 771;
columnValueArray[2].data = user.userAddressLine2;
columnValueArray[2].dataSize = sizeof(user.userAddressLine2);

columnValueArray[3].columnID = 772;
columnValueArray[3].data = user.userAddressLine3;
columnValueArray[3].dataSize = sizeof(user.userAddressLine3);

errCode = DbCopyColumnValues(dbRef, rowID,
   numColumns, columnValueArray);
if (errNone == errCode){
   // iterate through the column value array to check
   // for retrieval errors
   for (int i =0 ; i < numColumns; i++){
      // process the user name column
      // process each column value directly from the user
      // structure or from columnValueArray[i].data.
      if (errNone == columnValueArray[0].errCode)
         FldSetTextPtr(fldP, user.username);
      else {
         // handle error in retrieving column value.
         ErrDisplay("Error in retrieving column value");
         break;
      }
      // similarly, process the other columns…
   }
} else {
   ErrDisplay("Error in retrieving column values");
   return errCode;
}

// no storage heap buffer release required here as column
// values are retrieved in a user-allocated buffer
```

Writing Data

Just as you can read either a single column value or multiple column values, you can also write a single column value or multiple column values. `DbWriteColumnValue()` writes a single column value to the database. As when reading, for greater efficiency when writing more than one column value call `DbWriteColumnValues()` rather than calling `DbWriteColumnValue()` multiple times.

Partial column value writes are also possible for string, blob and vector columns through the use of an offset. This is useful for columns that contain large strings or blobs where, for space efficiency reasons, it makes sense to only write a portion of the column value.

When calling either of these `DbWrite...()` functions, the Data Manager copies the input data values to the storage heap as row data. Because the database now contains a copy of the data, you may then free the input data.

Listing 2.5 shows how to use `DbWriteColumnValue()` to write a single column value to a schema database.

Listing 2.5 Writing a single column value

```
uint32_t columnID = 1034;
char newName[] = “Terrence”;
uint32_t nameSize = strlen(newName) + 1;  // include the null
int32_t oldSize = -1;   // replace the entire column’s data

// this will overwrite old name with new name. Other
// variations are possible depending on
// combinations of bytesToReplace and srcBytes
```

```
if (errNone != DbWriteColumnValue(dbRef, &rowID,
   columnID, 0, oldSize, newName, nameSize)) {
   // handle error in writing column value.
   ErrDisplay("Error in writing column value");
}
```

Listing 2.6 shows how to use `DbWriteColumnValues()` to write multiple column values to a schema database.

Listing 2.6 Writing multiple column values

```
DbSchemaColumnValueType columnValueArray[3];
uint32_t columnIDArray[] = {1034, 1035, 1036};
uint32_t numColumns = sizeof(columnIDArray)/sizeof(uint32_t);
status_t errCode;

typedef struct {
   uint32_t orderID;
   char orderType[4];
   uint32_t orderQuantity;
} orderDetailsType;

orderDetailsType order;

columnValueArray[0].data = order.orderID;
columnValueArray[0].dataSize = sizeof(order.orderID);
columnValueArray[0].columnID = columnIDArray[0];

columnValueArray[1].data = order.orderType;
columnValueArray[1].dataSize = sizeof(order.orderType);
columnValueArray[1].columnID = columnIDArray[1];

columnValueArray[2].data = order.orderQuantity;
columnValueArray[2].dataSize = sizeof(order.orderQuantity);
columnValueArray[2].columnID = columnIDArray[2];

if (errNone != DbWriteColumnValues(dbRef, &rowID,
   numColumns, columnValueArray)){
   // handle error in writing column value.
   ErrDisplay("Error in writing column value");
}
```

Deleting Rows

Delete individual database rows by calling `DbDeleteRow()`. To delete a set of rows in a single table, create a cursor that identifies those rows and then call `DbCursorDeleteAllRows()`.

Cursors

Cursors simplify data access for schema databases. A cursor is a logical view of a subset of rows from a table, ordered as specified by the cursor. Once a cursor is created, applications can iterate the rows from the cursor, retrieve data from rows in the cursor, and to write data to rows in the cursor.

Cursors are temporary. They are not saved with the database. Cursors are simple to create and an application can have multiple cursors active at the same time, including multiple cursors on the same table.

With the exception of `DbInsertRow()`, schema database functions with row access semantics can take either a row ID or a cursor ID as a parameter. These are both `uint32_t` values and generally may be used interchangeably. The Data Manager derives the actual type of the parameter based on a value-encoding scheme it uses for row IDs; this scheme ensures that a row ID is always differentiable from a cursor ID. If you need to know whether a given identifier is a row ID or a cursor ID (or neither), you can make use of the functions `DbIsRowID()` and `DbIsCursorID()`.

The rows in a cursor needn't be sorted. A cursor that is opened unsorted is said to use the **default sort index**. In this instance, the string you supply for the *`sql`* parameter in the `DbCursorOpen...()` call should consist of the name of the table containing the database rows to be included and an optional WHERE clause indicating which of the table's rows should be included in the cursor. (See "The WHERE Clause" on page 41 for more information on the WHERE clause.)

Creating Cursors

Create a cursor with `DbCursorOpen()` or `DbCursorOpenWithCategory()`. To create a cursor you supply a reference to an open database; a SELECT statement that specifies the database table from which the rows are to be taken, an optional selection criteria (WHERE clause), and an optional sort criteria (ORDER BY clause); and a set of flags that indicate whether deleted or secret rows should be included in the cursor, whether the rows should be sorted by category, and so on. (See "Cursor Open Flags" on page 302 for the complete set). If you use `DbCursorOpenWithCategory()` you also can limit the rows in the cursor to those that meet the specified category criteria.

IMPORTANT: The sort index—that is, the SELECT statement—that you supply when creating the cursor must have been added to the table prior to its use in the `DbCursorOpen...` call. See the documentation for the `DbAddSortIndex()` function for more information.

The SELECT Statement

You use a limited form of the standard SQL SELECT statement to specify the rows that make up the cursor and the order in which those rows are to occur. You pass this SELECT statement, as an ASCII string, to `DbCursorOpen...()`. The following is the basic format of the schema database SELECT statement:

```
[SELECT * FROM] tableName [WHERE column op arg]
[ORDER BY (col1, col2, ...) [DESC | ASC | CASED | CASELESS]
[, col...]]
```

"SELECT * FROM" is entirely optional; its inclusion has no effect at this point: schema database cursors don't do projection. *tableName* is the only required part of this statement, and must identify the table from which the cursor rows are to be taken. The optional WHERE clause allows you to filter the rows to be included in the cursor; see "The WHERE Clause" on page 41 for a complete description of this clause.

The ORDER BY clause, also optional, controls the sorting of the rows within the cursor. Schema databases support two levels of sort keys, using parenthesis to identify the levels. The ORDER BY clause is perhaps best illustrated by way of example:

```
myTable ORDER BY LNAME, FNAME DESC, (34, 56) ASC CASED
```

The rows are sorted according to the column names and IDs as listed here. The first column ("LNAME", in the above example) gets the highest priority. The second column ("FNAME", in the above example) determines the order within duplicate values of the first. And the third column determines the order within duplicate values of the second. In this example the third column ID is a two-level key: column 34 is used unless that column is empty, in which case column 56 is used instead. DESC, ASC, and CASED are options that clarify how the sort is performed. The following options are allowed:

`DESC`
: (or `DESCENDING`): sort in descending order.

`ASC`
: (or `ASCENDING`): sort in ascending order. This is the default if neither `DESC` or `ASC` is specified.

`CASED`
: Take case into account when sorting.

`CASELESS`
: Ignore case when sorting. This is the default if neither `CASED` or `CASELESS` is specified.

Before you can use the SELECT statement when opening a cursor (other than one corresponding to the default sort index), you must have added to the database a sort index with a matching SELECT statement . This is done for efficiency reasons: schema databases maintain a list of rows in sorted order for each of the database's sort indices, and as a row is added, deleted, or modified the record lists for each sort index that applies to that row are updated. Because the lists are maintained in sorted order, the Data Manager doesn't have to perform a sort operation when you open a cursor that corresponds to an existing sort index.

Sort Indicies

Sort indices allow you to specify how table rows should be automatically sorted. These sort indices are maintained by the Data Manager and are stored as part of the database. Any application that has read authorization for a database can use the sort indices for that database. Any application that has write authorization for the database can add, remove, or edit the sort indices for a database.

There is no limit to the number of sort indices that you can define for a database, although for performance reasons you should limit the number of sort indices to a small number. Large numbers of sort indices affect the performance of adding, deleting, and modifying rows, because all indices must be adjusted appropriately as data in the database changes.

When creating a sort index, you use the format discussed under "The SELECT Statement" on page 37 to specify the table name and the keys (by column name or ID) that constitute the sort index. A sort index can sort on multiple keys; one of those keys is designated as the primary sort key. The other key specifications are optional and constitute the secondary sort keys.

Each key definition consists of the set of columns that constitute the key, the sort order (ascending or descending), and an indication as to whether or not row comparisons should be made in a case-sensitive manner. A key can be composed of multiple columns, although all of a key's columns must be of the same type. During a sort index update, when comparing two rows, if a row does not contain data in the first column of the sort key, the next specified column is checked and so on until a column with data is found. If the data in these two columns is equal, the next non-empty specified columns are checked.

The Data Manager uses its own internal sorting and comparison routines to keep the index automatically sorted. Whenever a field is updated, all indices (except the default index) that use that field are automatically updated.

Sort indices support the data types listed in Table 2.6. Only columns of the listed types may be used for the sort indices. For `dbChar` and `dbVarChar` data types, you can indicate whether or not a case-sensitive comparison should be performed. Note that the Data Manager relies upon the Text Manager comparison APIs when comparing these data types. This ensures correct sorting with the appropriate case-sensitivity on localized string data. (Data is sorted using the current system locale.) Blob data (`dbBlob`) is compared using a simple `memcmp()`.

Table 2.6 Data types supported by sort indices

`dbUInt8`	`dbUInt16`
`dbUInt32`	`dbUInt64`
`dbInt8`	`dbInt16`
`dbInt32`	`dbInt64`
`dbFloat`	`dbDouble`
`dbBoolean`	`dbDateTime`
`dbDate`	`dbTime`
`dbChar`	`dbVarChar`
`dbBlob`	

Application-provided comparison functions are not supported by sort indices, due to the performance overhead of having to call and potentially launch an application each time a field is modified.

Add a sort index to a database with `DbAddSortIndex()`. If you no longer need a particular sort index you can improve the efficiency of the database by removing it (so that the database no longer has to maintain a list of rows in sorted order for that sort index) by calling `DbRemoveSortIndex()`. Use the following functions to further manipulate the sort indices in a schema database:

`DbNumSortIndexes()`
: Get the number of sort indices defined for a given database. Within a database the defined sort indices have index values that range from 0 to one less than this number. Thus this function is particularly useful when iterating through a database's sort indices.

`DbGetSortDefinition()`
: Get a sort index given its position in the list of sort indices defined for a database.

`DbHasSortIndex()`
: Determine whether a particular sort index has been defined for a database. This function takes the same string that you

supply when adding a sort index to a database or opening a cursor.

When you no longer need a particular cursor, call `DbCursorClose()` to free all resources associated with the cursor.

An application can temporarily suspend automatic sorting of the currently opened database by calling `DbEnableSorting()` with the *`enable`* parameter set to `false`. This can be useful when doing a bulk update to the database, or during synchronization. Calling `DbEnableSorting()` with the *`enable`* parameter set to `true` will re-enable automatic sorting and causes the indices to be re-sorted.

The WHERE Clause

The Data Manager parses WHERE clauses and uses the information provided by applications to filter the set of rows returned as members of a cursor. For example, an application might request a cursor containing all rows where the value is greater then 42.

The general format of the WHERE clause is:

```
column_name_or_ID operator value
```

In an SQL string the WHERE clauses must come after the table name and before an ORDER BY clause if one is provided. A simple example is "`table WHERE AGE >= 42`"; the resulting cursor would only contain rows where the value of the column named "AGE" is greater than or equal to 42.

NOTE: Although the general format of the WHERE clause indicates that you can use a column ID in place of the column name, this may not be supported in future releases. Developers should use column names when specifying a WHERE clause.

Complex requests are supported by using the operators `AND` and `OR`. Both of these operators take WHERE clauses as their operands, allowing you to string requests together. `OR` has a lower operator precedence then `AND`, so all of the `AND` conditions are evaluated before the `OR` conditions. You can use parenthesis to group sub-clauses if operator precedence is an issue.

The `PS_LIKE` operator allows applications to perform sub-string matching. The operand is compared with the value in the requested column using the `TxtFindString()` function. Positive matches are added to the cursor, while non-matches are not.

The `IS NULL` and `IS NOT NULL` operators allow you to determine if a column has a value or is `NULL`. A `NULL` column value represents a lack of any value for a column. These operators may be used on all column types.

Table 2.7 lists the supported operators and the column types they can be used with.

Table 2.7 WHERE clause operators

Operator	Name	Supported Operand Types
=	Equal to	`dbBoolean, dbUInt32, dbInt32, dbUInt16, dbInt16, dbUInt8, dbInt8, dbDateTimeSecs, dbVarChar`
<>	Not equal to	`dbBoolean, dbUInt32, dbInt32, dbUInt16, dbInt16, dbUInt8, dbInt8, dbDateTimeSecs, dbVarChar`
<	Less than	`dbUInt32, dbInt32, dbUInt16, dbInt16, dbUInt8, dbInt8, dbDateTimeSecs, dbVarChar`
<=	Less than or equal to	`dbUInt32, dbInt32, dbUInt16, dbInt16, dbUInt8, dbInt8, dbDateTimeSecs, dbVarChar`

Table 2.7 WHERE clause operators *(continued)*

Operator	Name	Supported Operand Types
`>`	Greater than	`dbUInt32, dbInt32, dbUInt16, dbInt16, dbUInt8, dbInt8, dbDateTimeSecs, dbVarChar`
`>=`	Greater than or equal to	`dbUInt32, dbInt32, dbUInt16, dbInt16, dbUInt8, dbInt8, dbDateTimeSecs, dbVarChar`
`PS_LIKE`	PalmSource Like	`dbVarChar`
`AND`	And	Other WHERE clauses
`OR`	Or	Other WHERE clauses
`IS NULL`	Is NULL	All
`IS NOT NULL`	Is not NULL	All

Moving Through the Rows in a Cursor

When you create a cursor, the Data Manager takes a snapshot of the cursor's row IDs. This snapshot is used for iterating rows and is not affected by sorting updates. This is important to note, since operations that affect the number and order of rows in a database table won't affect the cursor contents until you explicitly refresh the cursor with `DbCursorRequery()`.

Cursors have a concept of a current row. When you open a cursor the current row is initially positioned at the first row. `DbCursorMove()` alters that current position: it can be used in a variety of ways. For convenience, the Data Manager includes a set of macros that simplify the process of altering the current row position:

`DbCursorMoveFirst()`
Moves the current row position to the first row in the cursor.

`DbCursorMoveLast()`
: Moves the current row position to the last row in the cursor.

`DbCursorMoveNext()`
: Moves the current row position one row forward.

`DbCursorMovePrev()`
: Moves the current row position one row backward.

`DbCursorMoveToRowID()`
: Move the current row position to the row with the specified ID.

`DbCursorSetAbsolutePosition()`
: Moves the current row position to the row with the specified index.

IMPORTANT: The first row in a cursor has an index value (position) of 1, similar to ODBC and JDBC. This differs from other aspects of schema database programming: the first column in a table has an index value of zero, and the first table in a database also has an index value of zero.

These macros, plus the fact that an error code is returned if you attempt to move beyond the bounds of the cursor, make it simple to iterate through a cursor's rows. See Listing 2.7 for an example of how to do this.

Listing 2.7 Iterating through a cursor's rows

```
status_t err;

err = DbCursorMoveFirst(myCursor);
if(err == errNone){
   while(!DbCursorIsEOF(myCursor)){
      // do something with the row data here, using the
      // cursor to indicate the current row. Like this:
      DbCopyColumnValue(dbRef, myCursor, ...);

      DbCursorMoveNext(myCursor);
   }
}
```

Because the various Data Manager functions that accept a row ID also accept a cursor ID, you needn't obtain the row ID of the current cursor row. As shown in the above example, just supply the cursor ID when calling a function such as `DbCopyColumnValue()`.

Rows that have been modified are not moved to their new sort position until `DbCursorRequery()` is called. Similarly, any newly-added rows are not available to the cursor until `DbCursorRequery()` is called. By calling `DbCursorRequery()`, you can refresh the cursor at any time to reflect the latest changes and sorting. Note that when a refresh occurs the current row may move to a new position and future move operations will move from the new position, not the old position. For example, if you change the data in the current row such that the row would wind up at the end of the cursor, and you then call `DbCursorRequery()`, a subsequent call to `DbCursorMoveNext()` will result in a `dmErrCursorEOF` error.

Data Variable Binding

Cursors allow you to bind variables to columns of the schema. When a variable is bound to a column, that variable is automatically updated with the field value of the current row in the cursor whenever the cursor's current position is changed. You needn't call `DbGetColumnValues()`; the data is automatically copied to the bound variables for you.

When calling `DbCursorBindData()` (or `DbCursorBindDataWithOffset()`), you must specify the ID of the column to which the variable is to be bound, a pointer to a data buffer (the bound variable), the length of that buffer, a pointer to a separate variable to hold the size of the data returned in the data buffer if the column type is one that has varying length, and a pointer to a variable that will receive an error code that is set each time the variable is updated. The error code will be set to `errNone` if the data is copied to the bound variable successfully, to `dmErrNoColumnData` if the column contains no data, or to some other value if an error of a different sort occurred.

The `DbCursorBindDataWithOffset()` function is similar to `DbCursorBindData()` but adds an extra parameter that lets you specify a byte offset into the field's data. The data copied to the variable is taken from the database field at the specified offset. This allows you to bind a subset of the field data to a variable.

You need to call `DbCursorBindData()` (or `DbCursorBindDataWithOffset()`) once for each column that you want to automatically retrieve or set data. It is not necessary to bind every column in the schema; only bind those that you are interested in. See Listing 2.8 for an example of how to use data variable binding.

Listing 2.8 Data variable binding example

```
uint32_t cursor;
char name[32];
char phone[24];
uint32_t sizeName;
uint32_t sizePhone;
status_t errName;
status_t errPhone;

dbRef = DbOpenDatabase(dbID, dmModeReadWrite, dbShareNone,
   idSortByName);

// Create the cursor
err = DbCursorOpen(dbRef, selectString, 0, &cursor);

// Bind the local variables to columns
DbCursorBindData(cursor, idColName, name, 32, &sizeName,
&errName);
DbCursorBindData(cursor, idColPhone, phone, 24, &sizePhone,
   &errPhone);

// Read and display all rows in the cursor
err = DbCursorMoveFirst(cursor);
while (err == errNone){
   // Data is now in bound variables, so display it
   DisplayNameAndPhone(name, sizeName, phone, sizePhone);

   // Get data for next row
   err = DbCursorMoveNext(cursor);
}

// Change the field values in the 5th row in cursor
DbCursorMoveTo(cursor, 4);
strcpy(name, "John Doe");
sizeName = strlen(name);
```

```
strcpy(phone, "555-1234");
sizePhone = strlen(phone);
err = DbCursorUpdate(cursor);

DbCursorClose(cursor);
```

Variable binding can also be used to write data to the database. Simply set each bound variable to its desired value, then call `DbCursorUpdate()`. All values are written to the database for the current row. Note that for varying-length types (`dbVarChar` and `dbBlob`) you should also set the corresponding `dataSize` variable—specified when you bound the variable to the schema column—to indicate the size of the data to be written back to that field.

NOTE: You must call `DbCursorUpdate()` each time you wish to update a schema database row with the contents of its bound variables. Changing the cursor's current position transfers data from the row to the bound variables; it doesn't automatically transfer data from the bound variables to the row's fields.

Secure Databases

Some applications need to create secure databases that restrict access to the database. The Data Manager supports the creation of secure databases that are protected by application-defined access rules, which are also known as **rule sets**.

To create a secure database, use `DbCreateSecureDatabase()`. When a secure database is initially created, it is completely protected and cannot be opened until access rules allowing read or write access have been defined for the database. `DbCreateSecureDatabase()` returns an initial rule set for the newly-created secure database. The initial rule set contains only one rule that allows the calling application, and no other, to modify the database's access rules. This is known as **modify access**.

There are six different actions that can be used in access rules:

`dbActionRead`

`dbActionWrite`

`dbActionDelete`

`dbActionSchemaEdit`

`dbActionBackup`

`dbActionRestore`

Access rules can require a digital signature, require a password, require a PIN, or allow unrestricted access. You can define different access rules for each of the different actions defined by the Data Manager. For example, a secure database could be configured to allow read access to anyone, but require a password for all other access. Creating access rules that require digital signatures provides for databases that can only be accessed by applications that have the correct digital signature. For more information about access rules, see *Exploring Palm OS: Security and Cryptography*.

Security is maintained at the database level, not for each individual row. All rows in the database have the same level of security. There is no way to assign different levels of security for different rows in the same database.

Secure databases are only visible to the Data Manager process. They are stored in the Data Manager's private secure storage heap, separate from unsecured databases. Applications can use the Data Manager catalog functions, such as `DmFindDatabase()`, to determine if the secure database exists. But the database data is not available to an application until the application, the user, or both have been authorized.

When an application requests access to a secure database, the Data Manager first calls the Authorization Manager to verify that the current user and/or application has rights to access the database. If the Authorization Manager approves access to the secure database, the Data Manager copies the requested rows to the application process as needed. For read operations the database data is copied from the Data Manager's private secure storage heap to the application's dynamic heap. Note since the data is copied to the application's dynamic heap, the data is writable. Even though it is writable, the application must still call the appropriate Data Manager write functions to update the data. Writing directly to the copy of the data in the dynamic heap has no effect on the row data in the database.

The Data Manager requires `dbActionRead` authorization when using the following functions on a secure database:

- `DbOpenDatabase()` with read-only mode
- `DbOpenDatabaseByTypeCreator()` with read-only mode

The Data Manager requires `dbActionWrite` authorization when using the following functions on a secure database:

- `DbOpenDatabase()` with write-only mode or read-write mode
- `DbOpenDatabaseByTypeCreator()` with write-only mode or read-write mode
- `DmSetDatabaseInfo()`
- `DmSetDatabaseProtection()`

The Data Manager requires `dbActionDelete` authorization when using the following functions on a secure database:

- `DmDeleteDatabase()`

The Data Manager requires `dbActionSchemaEdit` authorization when using following APIs on a secure database:

- `DbAddSchema()`
- `DbAddColumn()`
- `DbRemoveSchema()`
- `DbRemoveColumn()`
- `DbSetColumnPropertyValue()`
- `DbSetColumnPropertyValues()`
- `DbRemoveColumnProperty()`

The Data Manager requires `dbActionBackup` authorization when using the following functions on a secure database:

- `DmBackupInitialize()`
- `DmBackupUpdate()`
- `DmBackupFinalize()`

The Data Manager requires `dbActionRestore` authorization when using the following functions on a secure database:

- `DmRestoreInitialize()`

- `DmRestoreUpdate()`
- `DmRestoreFinalize()`

All other Data Manager functions do not require authorization when used on a secure database, because they either require a previous open call before they can be used, or they do not perform an operation that necessitates authorization.

Once a secure database has been successfully authorized and opened, the Data Manager places a special key in the calling application's process that indicates that the process has been authorized to use the database. All Data Manager functions that take a `DmOpenRef` as a parameter use this special key as proof that the application is allowed access. This allows the Data Manager to detect forged `DmOpenRef`s without needing to call the Authorization Manager for every function. The key is revoked when the database is closed.

The Data Manager also provides a function, `DbGetRuleSet()`, that allows an application to get the current rule set for a secure database. Once the rule set is obtained, the application can modify the access rules for the secure database—provided that the application has modify access. Once a secure database is open, any change in the access rules do not apply until the database is reopened.

Note that the Data Manager does not provide functions for creating or modifying the access rules, only functions for creating secure databases. Your application must use functions provided by the Authorization Manager and the Authentication Manager to define the access rules for a secure database.

Secure Databases and HotSync Operations

The Data Manager restricts access to a secure database to only those applications and users authorized by the database's access rules. During a sync operation the HotSync® client on the handheld uses Data Manager functions to access the handheld databases on behalf of the conduits running on the desktop. The HotSync client application must be able to access secure databases that need to be synchronized or backed up.

In order for an application to ensure that its secure database is syncable, it must modify the database access rules so that the HotSync client has special "bypass" access using the `AzmLibSetBypass()` function. When the HotSync client is given bypass access, any conduit on the desktop is able to access the database (the HotSync process does not provide a way to restrict access on a per-conduit basis). The bypass access must be made for each action needed. Since you can grant the HotSync client bypass access for each action separately, you can, for example, give the HotSync client read access, but not write or delete access.

If the HotSync client is not given bypass access, it is subject to the normal access rules as defined by the application. For example, if an application defines the access rules for its database so that only signed applications have access (read, write, or delete), during a HotSync operation the database isn't syncable since the HotSync client doesn't have the proper signature required to access the data. Therefore to allow syncing of the database the application must give "bypass" access to the HotSync client, which essentially grants access both to the HotSync client and to any properly-signed application.

The HotSync client on the handheld maintains a notion of trusted desktops. The HotSync process doesn't allow syncing or backing up of secure databases to non-trusted desktops.

Backing up Secure Databases

When a secure database is backed up to the desktop it is sent to the desktop in encrypted form and is saved on the desktop encrypted. During a backup operation the Data Manager encrypts the data. This differs from a sync operation; when data is sent to the desktop during synchronization it is sent "in the clear"—it is not encrypted.

Secure databases that were encrypted during backup can only be decrypted and restored by the Data Manager. The Data Manager provides special functions to perform the backup and restore operations: you use a combination of `DmBackupInitialize()`, `DmBackupUpdate()`, and `DmBackupFinalize()` to back the database up, and `DmRestoreInitialize()`, `DmRestoreUpdate()`, and `DmRestoreFinalize()` to restore the data. Note that these backup and restore functions work with both secure and non-secure databases.

Concurrent Database Access

When you open a non-schema database with write access, you have exclusive access to that database: no one else can open that database while you have it open, even if they are just opening it with read access. Or, when you open a non-schema database with read access, no one else can open that same database with write access. This can be somewhat restrictive: on a communicator-style device, for example, if you are editing a record in the address book when the phone rings, the phone application running in another process couldn’t open the address book in order to perform a caller-ID lookup.

Schema databases don’t have this problem because they support concurrent access to a single database. Note that schema databases don’t support concurrent write access: only one writer and multiple readers are allowed.

When opening a schema database you specify a **share mode** in addition to an access mode. The following share mode constants are supported for schema databases. Only one share mode can be specified when opening a database.

`dbShareNone`
No one else can open this database.

`dbShareRead`
Others can open this database with read access.

`dbShareReadWrite`
Others can open this database with read or write access.

Concurrent write access to the same database is not supported. That is, specifying an access mode of `dmModeReadWrite` and a share mode of `dbShareReadWrite` is not supported; an error will be returned if you attempt to open a database with this combination of access and share modes.

Table 2.8, below, shows all of the allowed combinations of access modes and share modes, and identifies which combinations can be used together (those that are marked “OK”).

Table 2.8 Allowable concurrent access/share mode combinations

	Mode=R Share=None	**Mode=R Share=R**	**Mode=R Share=R/W**	**Mode=R/W Share=None**	**Mode=R/W Share=R**
Mode=R Share=None	sharing not allowed	sharing not allowed	sharing not allowed	sharing not allowed	sharing not allowed
Mode=R Share=R	sharing not allowed	OK	OK	sharing not allowed	sharing not allowed
Mode=R Share=R/W	sharing not allowed	OK	OK	sharing not allowed	OK
Mode=R/W Share=None	sharing not allowed	sharing not allowed	sharing not allowed	sharing not allowed	sharing not allowed
Mode=R/W Share=R	sharing not allowed	sharing not allowed	OK	sharing not allowed	sharing not allowed

When sharing is enabled (that is, when the database is opened with shared read or shared read/write), the Data Manager server synchronizes access to the database. The synchronization is done at the database level. Each schema database function call is atomic, thus providing data integrity at the function level. Since the Data Manager doesn't support multiple applications writing to the same database, it doesn't have to deal with issues around concurrent updates.

As discussed in "Reading Data" on page 29, you can access record values by copy or by reference. When using the "by reference" functions to read record values from a database opened with shared write access, the Data Manager maintains a reference count of the number of active readers for each row. Applications can only modify a row if its reference count is 0—that is, if no one is currently reading that row. This protects the row against concurrent updates.

Whenever a schema database row is modified, added, deleted, or removed, the row index and any sort indices are automatically updated. This can only be done when the database is opened with write access. If another process has concurrently opened the same database with read access, however, it too will be affected by the changes to the sort indices. This is not a problem, however, since cursor shield the application from changes like this.

Working with Non-Schema Databases

Schema databases impose a structure upon the data, organizing it into tables, rows, and columns. Non-schema databases, on the other hand, impose less overhead and are significantly more flexible. Of course, your application generally has to do more work when dealing with non-schema databases, since your application is entirely responsible for interpreting the structure of each record.

Non-schema databases can either be record or resource databases. A **record database** holds application data. Each record can be structured in any fashion that the application desires. **Resource databases** are used to contain executable code, application resources, and the like.

In Palm OS Cobalt, non-schema databases come in two “flavors”: classic and extended. Classic databases are provided for compatibility with previous versions of Palm OS (and with applications running on Palm OS Cobalt through PACE). Because of a couple of long-standing limitations, however, unless your application needs this level of compatibility it should use extended or schema databases instead. Both classic and extended databases can be either record or resource databases.

Extended databases are very similar to classic databases. They have the following differences:

Classic Database	Extended Database
Records cannot exceed 64 KB in size.	Records can be more than 64 KB in length.
Are uniquely identified by name.	Are uniquely identified by a combination of name and creator ID.
Data should be stored in big-endian format (for 68K compatibility).	Data can be stored in either big-endian or little-endian format.

Because the two non-schema database types are so similar, you use many of the same functions when working with either database type. One of the most important functions that works only on extended databases is `DmCreateDatabase()`. To create a classic database, you use `DmCreateDatabaseV50()` instead. Other functions behave differently depending on whether you are operating on a classic or an extended database, and still others—such as `DmFindDatabase()`—use parameters to control their behavior in this area.

Structure of a Non-Schema Database Header

A non-schema database header consists of some basic database information and a list of records in the database. Each record entry in the header has the MemHandle of the record, 8 attribute bits, and a 3-byte unique ID for the record.

This section provides information about database headers, discussing these topics:

- Database Header Fields
- Structure of a Record Entry in a Non-Schema Database Header

IMPORTANT: Expect the database header structure to change in the future. Use the API to work with database structures.

Database Header Fields

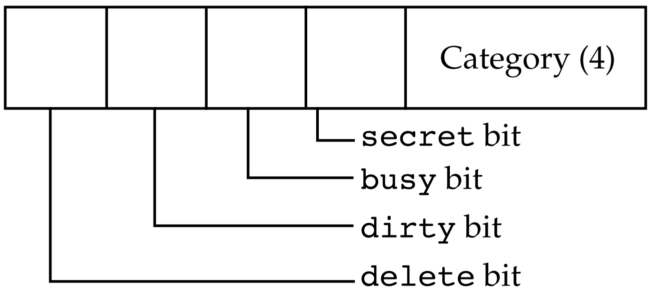
The database header has the following fields:

- The `name` field holds the name of the database.
- The `attributes` field has flags for the database.
- The `version` field holds an application-specific version number for that database.
- The `modificationNumber` is incremented every time a record in the database is deleted, added, or modified. Thus applications can quickly determine if a shared database has been modified by another process.
- The `appInfoID` is an optional field that an application can use to store application-specific information about the database. For example, it might be used to store user display preferences for a particular database.

- The `sortInfoID` is another optional field an application can use for storing the ID of a sort table for the database.
- The `type` and `creator` fields are each 4 bytes and hold the database type and creator. The system uses these fields to distinguish application databases from data databases and to associate data databases with the appropriate application.
- The `numRecords` field holds the number of record entries stored in the database header itself. If all the record entries cannot fit in the header, then `nextRecordList` identifies a `recordList` that contains the next set of records.

  Each record entry stored in a record list has three fields and is 8 bytes in length. Each entry has the MemHandle of the record which takes up 4 bytes: 1 byte of attributes and a 3-byte unique ID for the record. The `attribute` field, shown in Figure 2.1, is 8 bits long and contains 4 flags and a 4-bit category number. The category number is used to place records into user-defined categories like "business" or "personal."

Figure 2.1 Record Attributes



Structure of a Record Entry in a Non-Schema Database Header

Each record entry has the MemHandle of the record, 8 attribute bits, and a 3-byte unique ID for the record.

The unique ID must be unique for each record within a database. It remains the same for a particular record no matter how many times the record is modified. It is used during synchronization with the desktop to track records on the Palm Powered handheld with the same records on the desktop system.

The record attribute bits are set in the following circumstances:

- When the user deletes or archives a record the `delete` bit is set. Note, however, that its entry in the database header remains until the next synchronization with the PC.
- The `dirty` bit is set whenever a record is updated.
- The `busy` bit is set when an application currently has a record locked for reading or writing.
- The `secret` bit is set for records that should not be displayed before the user password has been entered on the handheld.

When a user "deletes" a record on a Palm Powered handheld, the record's data chunk is freed, the MemHandle stored in the record entry is set to 0, and the `delete` bit is set in the attributes. When the user archives a record, the deleted bit is also set but the chunk is not freed and the MemHandle is preserved. This way, the next time the user synchronizes with the desktop system, the desktop computer can quickly determine which records to delete (since their record entries are still around on the handheld). In the case of archived records, the conduit can save the record data on the desktop before it permanently removes the record entry and data from the handheld. For deleted records, the conduit just has to delete the same record from the desktop before permanently removing the record entry from the handheld.

Working with Non-Schema Databases

Using the Data Manager is similar to using a traditional file manager, except that the data is broken down into multiple records instead of being stored in one contiguous chunk. To create or delete a database, call `DmCreateDatabase()` (or, for classic databases, `DmCreateDatabaseV50()`) and `DmDeleteDatabase()`.

To open a database for reading or writing, you must first get the database ID. Calling `DmFindDatabase()` searches for a database by name and type (schema, extended, or classic) and returns its database ID.

After determining the database ID, you can open the database for read-only or read/write access. When you open a database, the system locks down the database header and returns a reference to a database access structure, which tracks information about the open database and caches certain information for optimum performance. The database access structure is a relatively small structure (less than 100 bytes) allocated in the dynamic heap that is disposed of when the database is closed.

Call `DmDatabaseInfo()`, `DmSetDatabaseInfo()`, and `DmDatabaseSize()` to query or set information about a database, such as its name, size, creation and modification dates, attributes, type, and creator.

Call `DmGetRecord()`, `DmQueryRecord()`, and `DmReleaseRecord()` when viewing or updating a database.

- `DmGetRecord()` takes a record index as a parameter, marks the record busy, and returns a handle to the record. If a record is already busy when `DmGetRecord()` is called, an error is returned.
- `DmQueryRecord()` is faster if the application only needs to view the record; it doesn't check or set the busy bit, so it's not necessary to call `DmReleaseRecord()` when finished viewing the record.
- `DmReleaseRecord()` clears the `busy` bit, and updates the modification number of the database and marks the record dirty if the `dirty` parameter is true.

To resize a record to grow or shrink its contents, call `DmResizeRecord()`. During reallocation, the handle to the record may change. `DmResizeRecord()` returns the new handle to the record.

To add a new record to a database, call `DmNewRecord()`. This function can insert the new record at any index position, append it to the end, or replace an existing record by index. It returns a handle to the new record.

There are three methods for removing a record: `DmRemoveRecord()`, `DmDeleteRecord()`, and `DmArchiveRecord()`.

- `DmRemoveRecord()` removes the record's entry from the database header and disposes of the record data.
- `DmDeleteRecord()` also disposes of the record data, but instead of removing the record's entry from the database

header, it sets the deleted bit in the record entry attributes field and clears the local chunk ID.

- `DmArchiveRecord()` does not dispose of the record's data; it just sets the deleted bit in the record entry.

Both `DmDeleteRecord()` and `DmArchiveRecord()` are useful for synchronizing information with a desktop computer. Since the unique ID of the deleted or archived record is still kept in the database header, the desktop computer can perform the necessary operations on its own copy of the database before permanently removing the record from the Palm OS database.

Call `DmGetRecordAttr()`, `DmGetRecordCategory()`, and `DmGetRecordID()` to retrieve the record information stored in the database header, and `DmSetRecordAttr()`, `DmSetRecordCategory()`, and `DmSetRecordID()` to set this information. Typically, applications set or retrieve the category of a record, which is stored in the lower four bits of the record's attribute field.

To move records from one index to another or from one database to another, call `DmMoveRecord()`, `DmAttachRecord()`, and `DmDetachRecord()`. `DmDetachRecord()` removes a record entry from the database header and returns the record handle. Given the handle of a new record, `DmAttachRecord()` inserts or appends that new record to a database or replaces an existing record with the new record. `DmMoveRecord()` is an optimized way to move a record from one index to another in the same database.

Record Attributes

Table 2.4 lists the functions that you use to get and set a non-schema database record's ID, category, and attributes.

Table 2.9 Functions used to access record information

	Non-Schema Database

Local ID	`DmGetRecordID()`
	`DmSetRecordID()`

Table 2.9 Functions used to access record information

	Non-Schema Database
Category Membership	`DmGetRecordCategory()` `DmSetRecordCategory()`
Attributes	`DmGetRecordAttr()` `DmSetRecordAttr()`

Resource Databases

Structure of a Resource Database Header

A resource database header consists of some general database information followed by a list of resources in the database. The first portion of the header is identical in structure to a normal database header (see "Structure of a Non-Schema Database Header" on page 55). Resource database headers are distinguished from normal database headers by the `dmHdrAttrResDB` bit in the `attributes` field.

IMPORTANT: Expect the resource database header structure to change in the future. Use the API to work with resource database structures.

- The `name` field holds the name of the resource database.
- The `attributes` field has flags for the database and always has the `dmHdrAttrResDB` bit set.
- The `modificationNumber` is incremented every time a resource in the database is deleted, added, or modified. Thus, applications can quickly determine if a shared resource database has been modified by another process.
- The `appInfoID` and `sortInfoID` fields are not normally needed for a resource database but are included to match the structure of a regular database. An application may optionally use these fields for its own purposes.
- The `type` and `creator` fields hold 4-byte signatures of the database `type` and `creator` as defined by the application that created the database.

- The `numResources` field holds the number of resource info entries that are stored in the header itself. In most cases, this is the total number of resources. If all the resource info entries cannot fit in the header, however, then `nextResourceList` has the `chunkID` of a `resourceList` that contains the next set of resource info entries.

Each 10-byte resource info entry in the header has the resource type, the resource ID, and the ID of the Memory Manager chunk that contains the resource data.

Working with Resource Databases

You can create, delete, open, and close resource databases with the functions used to create normal record-based databases (see "Working with Non-Schema Databases" on page 54). This includes all database-level (not record-level) functions in the Data Manager such as `DmCreateDatabase()`, `DmDeleteDatabase()`, `DmDatabaseInfo()`, and so on.

When you create a new database using `DmCreateDatabase()`, the type of database created (record or resource) depends on the value of the `resDB` parameter. If set, a resource database is created and the `dmHdrAttrResDB` bit is set in the `attributes` field of the database header. Given a database header ID, an application can determine which type of database it is by calling `DmDatabaseInfo()` and examining the `dmHdrAttrResDB` bit in the returned `attributes` field.

Once a resource database has been opened, an application can read and manipulate its resources by using the resource-based access functions of the Resource Manager. Generally, applications use the `DmGetResource()` and `DmReleaseResource()` functions.

`DmGetResource()` searches a specified resource database and returns a handle to a resource, given the resource type and ID.

NOTE: Previous versions of Palm OS had the notion of a resource “search chain”, the set of all open resource databases that were searched when looking for a specified resource. This concept isn’t really supported in Palm OS 6, except for compatibility purposes: applications that run under PACE will work as originally designed. To support this level of compatibility, the Data Manager contains a number of deprecated functions that provide the old functionality. These functions are: `DmOpenDatabaseV50()`, `DmOpenDBNoOverlayV50()`, `DmOpenDatabaseByTypeCreatorV50()`, `DmGetResourceV50()`, and `DmGet1ResourceV50()`. Because these functions are deprecated, applications written for Palm OS 6 should not rely upon them.

`DmReleaseResource()` should be called as soon as an application finishes reading or writing the resource data. To resize a resource, call `DmResizeResource()`, which accepts a handle to a resource and reallocates the resource. It returns the handle of the resource, which might have been changed.

The remaining Resource Manager functions are usually not required for most applications. These include functions to get and set resource attributes, move resources from one database to another, get resources by index, and create new resources. Most of these functions reference resources by index to optimize performance. When referencing a resource by index, the `DmOpenRef` of the open resource database that the resource belongs to must also be specified. Call `DmSearchResourceOpenDatabases()` to find a resource by type and ID or by pointer by searching in all open resource databases opened by the process. Note that this function does not search resource databases opened in other processes.

To get the `DmOpenRef` of the topmost open resource database, call `DmNextOpenResDatabase()` and pass `NULL` as the current `DmOpenRef`. To find out the `DmOpenRef` of each successive database, call `DmNextOpenResDatabase()` repeatedly with each successive `DmOpenRef`.

Given the access pointer of a specific open resource database, `DmFindResource()` can be used to return the index of a resource, given its type and ID. `DmFindResourceType()` can be used to get the index of every resource of a given type. To get a resource handle by index, call `DmGetResourceByIndex()`.

To determine how many resources are in a given database, call `DmNumResources()`. To get and set attributes of a resource including its type and ID, call `DmResourceInfo()` and `DmSetResourceInfo()`. To attach an existing data chunk to a resource database as a new resource, call `DmAttachResource()`. To detach a resource from a database, call `DmDetachResource()`.

To create a new resource, call `DmNewResource()` and pass the desired size, type, and ID of the new resource. To delete a resource, call `DmRemoveResource()`. Removing a resource disposes of its data chunk and removes its entry from the database header.

Overlays

Resource databases (and only resource databases) can have overlay databases associated with them; these localization overlays provide a method of localizing a software module without requiring a recompile or modification of the software. Each overlay database is a separate resource database that provides an appropriately-localized set of resources for a single software module (the **base database**) and a single target locale (language and country).

When a resource database is opened, the Data Manager looks for an overlay matching the base database and the current locale. When searching for an overlay database, the Data Manager first looks in RAM. If an appropriate overlay database isn't found there for the specified base database and target locale, it then tries to locate one in ROM.

Most of the locale APIs are declared in the Locale Manager, which is documented in *Exploring Palm OS: Text and Localization*. The Data Manager does provide a few functions, however, that let you get and set the locale that is used when opening an overlay, that determines an overlay database's locale, and that identifies the proper overlay database given the name of a base database and a locale.

NOTE: There is no system support for letting the user pick the language of a given application. A separate application—the "language picker"—lets the user change the Data Manager's overlay locale. This application sets the overlay locale indirectly, by changing the system locale and thus forcing a soft reset.

The Data Manager's overlay locale is a global setting that applies to all processes and threads. The overlay locale is initialized to be the same as the system locale following a soft reset. After the overlay locale is changed by calling `DmSetOverlayLocale()`, whenever the Data Manager needs to automatically open an overlay it uses the specified locale. If no valid overlay exists for that overlay locale, the Data Manager uses the fallback overlay locale instead.

You set the Data Manager's overlay locale with `DmSetOverlayLocale()`, and you get it with `DmGetOverlayLocale()`. Similarly, set the fallback overlay locale by calling `DmSetFallbackOverlayLocale()` and get it by calling `DmGetFallbackOverlayLocale()`.

For a given overlay database, you can determine its locale by passing the overlay database name and a pointer to an `LmLocaleType` structure to `DmGetOverlayDatabaseLocale()`. Upon return, the `LmLocaleType` structure contains the overlay database's locale.

To locate the overlay database for a given base database, pass the name of the base database and an `LmLocaleType` structure indicating the desired locale to `DmGetOverlayDatabaseName()`. It will return the name of the overlay database for the specified base database and locale. You can pass `NULL` instead of a pointer to an `LmLocaleType` structure to obtain the overlay database name for the base database and the current locale.

Overlay Signature Verification

If the base database is signed, then the overlay database must also be signed, and its signature must be validated using a certificate ID that comes from the base database's `'sign'` resource. More specifically,

- The base database's `'sign'` resource must contain one or more overlay certificate ID values.

- The overlay database must contain a `'sign'` resource.
- One of the signatures in the overlay database's `'sign'` resource must use a certificate ID that comes from the base database's `'sign'` resource list of overlay certificate ID values, and this signature must validate the overlay database.

Data Manager Tips

Working properly with databases makes your application run faster and synchronize without problems. Follow these suggestions:

- Database names can be up to 31 characters in length, and on the handheld can be composed of any valid 7-bit ASCII characters (only). Some conduits—such as PalmSource's backup conduit—use a name-mangling scheme to preserve case-sensitive database names when generating backup filenames on Microsoft Windows. Other conduits may not do this, however, so you may want to avoid filenames that depend on case for distinction.

IMPORTANT: Previous versions of Palm OS didn't enforce the requirement that database names be composed only of 7-bit ASCII characters. Palm OS Cobalt requires that this be so.

  By convention, filename extensions are not used on the handheld. Instead, database types are used to identify databases as members of a certain type or class. Note that when the PalmSource backup conduit transfers a file to the desktop, it automatically appends one of the following extensions to the database filename:

  - PRC for resource databases (classic or extended)
  - PDB for non-schema record databases (classic or extended)
  - SDB for non-secure schema databases
  - SSD for secure schema databases

  - VLT for vault databases used to hold security information (HEKs, rules, tokens, and the like)

  The extension is removed when the file is transferred back to the handheld.

- When the user deletes a record from a database, call `DmDeleteRecord()` (or `DbDeleteRow()`) to remove all data from the record, not `DmRemoveRecord()` (or `DbRemoveRow()`) to remove the record itself. That way, the desktop application can retrieve the information that the record is deleted the next time there is a HotSync operation.

  Note: If your application doesn't have an associated conduit, call `DmRemoveRecord()` to completely remove the record.

- Keep data in database records compact. To avoid performance problems, Palm OS databases are not compressed, but all data are tightly packed. This pays off for storage and during HotSync operations.
- All records in a non-schema database should be of the same type and format. This is not a requirement, but is highly recommended to avoid processing overhead.
- Be sure your application modifies the flags in the database header appropriately when the user deletes or otherwise modifies information. This flag modification is only required if you're synchronizing with the PalmSource PIM applications, but should likely be done with any database that is to be sync'd by a conduit.
- Don't display deleted records.
- Call `DmSetDatabaseInfo()` when creating a non-schema database to assign a version number to your application. Databases default to version 0 if the version isn't explicitly set.
- Call `DmDatabaseInfo()` to check the non-schema database version at application start-up.

File Streaming Layer

The file streaming functions add a layer on top of the classic database functions and let you work with a Palm OS database using a more familiar set of operations. File streams allow you to read, write, seek to a specified offset, truncate, and do everything else you'd expect to do with a desktop-style file.

Other than backup and restore, Palm OS does not provide direct HotSync support for file streams.

The use of double-buffering imposes a performance penalty on file streams that may make them unsuitable for certain applications. Record-intensive applications tend to obtain better performance from the Data Manager.

Using the File Streaming API

The File Streaming API is derived from the C programming language's `<stdio.h>` interface. Any C book that explains the `<stdio.h>` interface should serve as a suitable introduction to the concepts underlying the Palm OS File Streaming API. This section provides only a brief overview of the most commonly used file streaming functions.

The `FileOpen()` function opens or creates a file (an extended database; use `FileOpenV50()` to open or create a classic database), and the `FileRead()` function reads it. The semantics of `FileRead()` and `FileWrite()` are just like their `<stdio.h>` equivalents, the `fread()` and `fwrite()` functions. The other `<stdio.h>` functions have obvious analogs in the File Streaming API as well.

For example,

```
theStream = FileOpen("KillerAppDataFile", 'KILR',
   'KILD', fileModeReadOnly, &err);
```

As on a desktop, the filename is the unique item. The creator ID and file type are for informational purposes and your code may require that an opened file have the correct type and creator.

IMPORTANT: Previous versions of Palm OS didn't enforce the requirement that database names passed to `FileOpen()` be composed only of 7-bit ASCII characters. Palm OS Cobalt requires that this be so.

Normally, the FileOpen() function returns an error when it attempts to open or replace an existing stream having a type and creator that do not match those specified. To suppress this error, pass the `fileModeAnyTypeCreator` selector as a flag in the `openMode` parameter to the FileOpen() function.

To read data, use the FileRead() function as in the following example:

```
FileRead(theStream, &buf, objSize, numObjs, &err);
```

To free the memory used to store stream data as the data is read, you can use the FileControl() function to switch the stream to destructive read mode. This mode is useful for manipulating temporary data; for example, destructive read mode would be ideal for adding the objects in a large data stream to a database when sufficient memory for duplicating the entire file stream is not available. You can switch a stream to destructive read mode by passing the `fileOpDestructiveReadMode` selector as the value of the `op` parameter to the FileControl() function.

The FileDmRead() function can read data directly into a Data Manager chunk for immediate addition to a Palm OS database.

3

Virtual File Systems

VFS Manager

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

NOTE: Although the great majority of the functions in the VFS Manager can be used by any application, some are intended only for use by drivers and file systems. Others are not intended for use by third-party applications but are designed primarily for system use.

The VFS Manager, the Data Manager, and File Streaming APIs

With the addition of the VFS Manager to Palm OS®, there are now three distinct ways applications can store and retrieve Palm OS user data:

- The Data Manager manages user data in the storage heap. Use them to store and retrieve Palm OS user data when storage on the handheld is all that is needed, or when efficient access to data is paramount.
- The File Streaming API is a layer on top of the Data Manager that provides file functionality with all data being read from or written to a database in the storage heap. Most applications have no need for the File Streaming APIs; they are primarily used by applications that need to work with large blocks of data.

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

Palm OS applications should use the appropriate APIs for each given situation. The Data Manager, being an efficient manager of storage in the storage heap, should be used whenever access to external media is not absolutely needed. Use the VFS API when interoperability and file system access is needed.

For more information on the Data and Resource Managers, as well as on the File Streaming APIs, see Chapter 2, "Palm OS Databases." For details of the APIs presented by the VFS Manager, see Chapter 8, "VFS Manager."

Checking for the Presence of the VFS Manager

Because not every system has (or needs) Virtual File System (VFS) Manager services, applications wishing to use these services should check to make sure they are present before calling them. This is accomplished by checking for the VFS Manager's system feature with a call to `FtrGet()`, supplying `sysFileCVFSMgr` for the feature creator and `vfsFtrIDVersion` for the feature number.

The following code shows how to check for the presence and proper version of the VFS Manager. Note that `expectedVFSMgrVersionNum` should be replaced by the actual version number you expect.

```
uint32_t vfsMgrVersion;
Err err;
err = FtrGet(sysFileCVFSMgr, vfsFtrIDVersion,
   &vfsMgrVersion);
if(err){
   // VFS Manager not installed
} else {
   // check version number of VFS Manager, if necessary
   if(vfsMgrVersion == expectedVFSMgrVersionNum)
      // everything is OK
}
```

Standard Directories

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

The following table lists the standard directory layout for all "standards compliant" Palm OS secondary storage. All Palm OS relevant data should be in the `/PALM` directory (or in a subdirectory of the `/PALM` directory), effectively partitioning off a private name space.

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

The Palm OS Launcher is expansion card aware. When an expansion card containing a file system is inserted, all applications listed in the card's `/PALM/Launcher` directory are automatically added to a new Launcher category. This new category takes the name of the expansion card volume. Note that the name displayed in the Launcher for a given application is the name in the application's `tAIN` (application icon name) resource or, if this resource is empty, the database name, which may or may not match the name of the file.

NOTE: Whenever possible give the same name to the `.prc` file and to the database. If the `.prc` filename differs from the database name, and users copy your application from the card to the handheld and then to another card, the filename may change. This is because the database name is used when an application is copied from the handheld to the card.

When a writable volume is mounted, the Launcher automatically creates the `/PALM` and `/PALM/Launcher` directories if they don't already exist. If they do, and if there are applications present in the `/PALM/Launcher` directory, the Launcher automatically switches to the card's list of applications unless it runs `start.prc`.

In addition to these standard directories, the VFS Manager supports the concept of a **default directory**; a directory in which data of a particular type is typically stored. See "Determining the Default Directory for a Particular File Type" on page 89 for more information.

Applications on Cards

Palm OS applications located in the `/PALM/Launcher` directory of an expansion card volume appear in a separate Launcher category when the card is inserted into the handheld's expansion slot. If you tap the icon for one of these applications, it is copied to main memory and then launched.

Applications launched from a card ("card-launched" applications) are first sent a `sysAppLaunchCmdCardLaunch` launch code, along with a parameter block that includes the reference number of the volume on which the application resides and the complete path to the application. When processing this launch code, the application shouldn't interact with the user or access globals. Unless the application sets the `sysAppLaunchStartFlagNoUISwitch` bit in the `start` flags (which are part of the parameter block), the application is then sent a `sysAppLaunchCmdNormalLaunch` launch code. This is when the application should, if it needs to, interact with user. Applications may want to save some state when `sysAppLaunchCmdCardLaunch` is received, then act upon that state information when `sysAppLaunchCmdNormalLaunch` is received.

When the user switches to a new application, the card-launched application is removed from main memory. Note, however, that any databases created by the card-launched application remain.

There are certain implications to this “copy and run” process:

- There must be sufficient memory for the application. If the handheld doesn’t have enough memory to receive the application, it isn’t copied from the expansion card and it isn’t launched.
- The copying process takes time. For large applications, this can cause a noticeable delay before the application is actually launched.
- If some version of the application on the card is already present in main memory, the Launcher puts up a dialog that requires the user to choose whether or not to overwrite the in-memory version.
- Card-launched applications have a limited lifetime: applications reside in main memory only while they are running. When the user switches to a different application, the card-launched application that was just running is removed from main memory. If the card-launched application is then re-launched, it is once again copied into the handheld’s memory.
- “Legacy” applications—those that are unaware that they are being launched from a card—only work with databases in main memory. Associated databases aren’t copied to main memory along with the application unless the database is bundled with the application. Databases created by card-launched applications are not removed along with the application, however, so this data is available to the application when it is subsequently run. Applications that are written to take advantage of the VFS Manager can read

and write data on the expansion card, so this limitation generally only applies to legacy applications.

Bundled databases, although copied to main memory along with their associated application, are meant for static data that doesn't change, such as a game level database. Bundled databases are not copied back to the card; they are simply deleted from memory when the user chooses another application. To bundle a database with an application, give it the same creator ID as the owning application, set the `dmHdrAttrBundle` bit, and place it in the `/PALM/Launcher` directory along with the application.

- Unless a card-launched application is running, it doesn't receive notifications or launch codes since it isn't present on the handheld. In particular, these applications don't receive notifications and aren't informed when an alarm is triggered.

Volume Operations

If an expansion card supports a file system, the VFS Manager allows you to perform a number of standard volume operations. To determine which volumes are currently mounted and available, use `VFSVolumeEnumerate()`. This function, the use of which is illustrated in "Checking for Mounted Volumes" on page 67 of *Exploring Palm OS: System Management* returns a volume reference number that you then supply to the remainder of the volume operations.

When the user inserts a card containing a mountable volume into a slot (note that the current implementation only supports one volume per slot), the VFS Manager attempts to mount the volume automatically. You should rarely, if ever, have to mount volumes directly. You can attempt to mount a volume using a different file system, however, perhaps after installing a new file system driver on the handheld. To explicitly mount or unmount a volume, use `VFSVolumeMount()` and `VFSVolumeUnmount`. When mounting a volume, you can either specify an explicit file system with which to mount the volume, or you can request that the VFS Manager try to determine the appropriate file system. If the VFS Manager cannot mount the volume using any of the available file systems, it attempts to format the volume using a file system deemed appropriate for the slot, and then mount it. See the description of `VFSVolumeMount()` in Chapter 8, "VFS Manager," for the precise arguments you must supply when explicitly mounting a volume.

Use `VFSVolumeFormat()` to format a volume. This function can be used to change the file system on the expansion card; you can explicitly indicate a file system to use when formatting it. Once the card has been formatted, the VFS Manager automatically mounts it; a new volume reference number is returned from `VFSVolumeFormat()`.

The `VFSVolumeGetLabel()` and `VFSVolumeSetLabel()` functions get and set the volume label, respectively. Since the file system is responsible for verifying the validity of strings, you can try to set the volume label to any desired value. If the file system doesn't natively support the name given, the VFS Manager creates the `/VOLUME.NAM` file used to support long volume names (see "Naming Volumes" on page 77 for more information) or you get an error back if the file system doesn't support the supplied string.

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

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

All of the above information is returned encapsulated within a `VolumeInfoType` structure. Whether the volume is hidden or read-only is further encoded into a single field within this structure;

see Volume Attributes in Chapter 8, “VFS Manager,” for the bits that make up this field.

Hidden Volumes

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

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

Matching Volumes to Slots

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

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

Although block device drivers currently only support one volume per slot, obtaining volume information that corresponds to a given slot reference number isn’t quite so simple, since there isn’t a function that returns the volume reference number given a slot reference number. You can, however, iterate through the mounted volumes and check each volume’s slot reference number. This is the technique that the CardInfo application uses.

Naming Volumes

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

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

When the underlying file system doesn't support a long volume name, `VFSVolumeSetLabel()` creates the file `/VOLUME.NAM` in an effort to preserve the long volume name. This file contains the following, in order:

Field	Description
`Char cookie[4]`	4-byte cookie that identifies this file. The value of this cookie is `vfsVolumeNameFileCookie`.
`UInt16 cacheLen`	Big-endian length, in bytes, of the cached file-system-level volume label.

Field	Description
`Char cacheLabel[cacheLen]`	Unicode UCS-2 format string containing the volume label as it is stored in the file system layer. This is compared with the file system volume label to see if the user has changed the volume label on a device that doesn't support the `/VOLUME.NAM` file. In this event, the file system volume label is used; the contents of `/VOLUME.NAM` are ignored.
`UInt16 length`	Big-endian length, in bytes, of the long volume label.
`Char label[length]`	Unicode UCS-2 format string containing the long volume label.

File Operations

Most of the familiar operations you'd use to operate on files in a desktop application are supported by the VFS Manager; these are listed in "Common Operations," below. In addition, the VFS Manager includes a set of functions that simplify the way you work with files that represent Palm OS databases (`.pdb`) or Palm resource databases (`.prc`). These are covered in "Working with Palm OS Databases" on page 81.

Common Operations

The VFS Manager provides many standard file operations that should be familiar from desktop and larger computer systems. Because these functions work largely as you would expect, their use isn't detailed here. See the descriptions of each individual function in Chapter 8, “VFS Manager,” for the arguments, return values, and side effects of each.

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

Table 3.1 Common file operations

Function	Description
`VFSFileOpen()`	Open a file, given a volume reference number and a file path.
`VFSFileClose()`	Close an open file.
`VFSFileRead()`	Read data from a file into the dynamic heap or any writable memory.
`VFSFileReadData()`	Read data from a file into a chunk of memory in the storage heap.
`VFSFileWrite()`	Write data to an open file.
`VFSFileSeek()`	Set the position within an open file from which to read or write.
`VFSFileTell()`	Get the current position of the file pointer within an open file.
`VFSFileEOF()`	Get the end-of-file status for an open file.
`VFSFileCreate()`	Create a file, given a volume reference number and a file path.
`VFSFileDelete()`	Delete a closed file.
`VFSFileRename()`	Rename a closed file.
`VFSFileSize()`	Obtain the size of an open file.
`VFSFileResize()`	Change the size of an open file.
`VFSFileGetAttributes()`	Obtain the attributes of an open file, including hidden, read-only, system, and archive bits. See “File and Directory Attributes” in Chapter 8, “VFS Manager,” for the bits that make up the attributes field.

Table 3.1 Common file operations *(continued)*

Function	Description
`VFSFileSetAttributes()`	Set the attributes of an open file, including hidden, read-only, system, and archive bits.
`VFSFileGetDate()`	Get the created, modified, and last accessed dates for an open file.
`VFSFileSetDate()`	Set the created, modified, and last accessed dates for an open file.

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

Naming Files

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

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

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

- The name is created from the first six valid, non-space characters which appear before the last period. The only valid characters are A-Z, 0-9, $, %, ‘, -, _, @, ~, ‘, !, (,), ^, #, and &.

- The extension is the first three valid characters after the last period.
- The end of the six byte name has “~1” appended to it for the first occurrence of the shortened filename. Each subsequent occurrence uses the next unique number, so the second occurrence would have “~2” appended, and so on.

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

Working with Palm OS Databases

Expansion cards are often used to hold Palm OS applications and data. Due to the way that secondary storage media are connected to the Palm Powered handheld, applications cannot be run directly from the expansion card, nor can databases be manipulated using the Data Manager without first transferring them to main memory. Applications written to use the VFS Manager, however, can operate directly on files located on an expansion card.

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

Stand-Alone Applications

To allow the user to run an application that is self-contained—that isn’t accompanied by a separate database—you need only do one of two things:

- If the application is to be run whenever the card is inserted into the expansion slot, simply name the application `start.prc` and place it in the `/PALM` directory. The operating system takes care of transferring the application to main memory and starting it automatically.

- If the application is to be run on-demand, place it in the `/PALM/Launcher` directory. All applications located in this directory appear in the launcher when the user selects the category bearing the name of the expansion card.

Both of these mechanisms allow applications that were written without any knowledge of the VFS or Expansion Manager APIs to be run from a card. Because they are transferred to main memory prior to being run, such applications need not know that they are being run from an expansion card. Databases created by these applications are placed in the storage heap, as usual. When the card containing the application is removed, the application disappears from main memory unless it is running, in which case it remains until such time as the application is no longer running. Any databases it created remain. When the card is re-inserted and the application re-run, it is once again copied into main memory and is able to access those databases.

Applications with Static Data

Many applications are accompanied by one or more associated Palm OS databases when installed. These applications, at least to a limited degree, need to be cognizant of the fact that they reside on an expansion card.

If there is no specific requirement for the application's data to be stored in Palm OS database format, you may want to use the VFS Manager's many file I/O operations to read and write the data on the card. Because of the large data storage capabilities of the expansion media relative to the handheld's memory, this latter solution is the one preferred by applications where large capacity data storage is a key feature.

Bundled Databases

When an application is launched from a card using the launcher, any bundled databases present in the `/PALM/Launcher` directory are also imported. Bundled databases have the same creator as the "owning" application and have the `dmHdrAttrBundle` bit set. Note that bundled databases are intended only for read-only data, such as a game-level database. Bundled databases are removed from main memory along with the application when the user switches to another application and are not copied back to the expansion card.

Transferring Palm OS Databases to and from Expansion Cards

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

`VFSExportDatabaseToFile()` and `VFSImportDatabaseFromFile()`, depending on the size of the database and the mechanism by which it is being transferred, can take some time. Use `VFSExportDatabaseToFileCustom()` and `VFSImportDatabaseFromFileCustom()` if you want to display a progress dialog or allow the user to cancel the operation. These functions make repeated calls to a callback function that you specify; within this callback function you can update a progress indicator. The return value from your callback determines whether the database transfer should proceed; return `errNone` if it should continue, or return any other value to abort the process. See the documentation for `VFSExportProcPtr()` and `VFSImportProcPtr()` in Chapter 8, "VFS Manager," for the format of each callback function.

The following code excerpt illustrates the use of `VFSImportDatabaseFromFileCustom()` with a progress tracker.

Listing 3.1 Using VFSImportDatabaseFromFileCustom()

```
typedef struct {
   ProgressType *progressP;
   const Char   *nameP;
} CBDataType, *CBDataPtr;

static Boolean ProgressTextCB(PrgCallbackDataPtr cbP) {
   const Char *nameP = ((CBDataPtr) cbP->userDataP)->nameP;
   // Set up the progress text to be displayed
   StrPrintF(cbP->textP, "Importing %s.", nameP);
   cbP->textChanged = true;

   return true;  // So what we specify here is used to update the dialog
}

static Err CopyProgressCB(UInt32 size, UInt32 offset, void *userDataP) {
   CBDataPtr CBDataP = (CBDataPtr) userDataP;

   if (offset == 0) {  // If we're just starting, we need to set up the dialog
      CBDataP->progressP = PrgStartDialog("Importing Database", ProgressTextCB,
         CBDataP);

      if (!CBDataP->progressP)
         return memErrNotEnoughSpace;
   } else {
      EventType event;
      Boolean   handled;

      do {
         EvtGetEvent(&event, 0);  // Check for events

         handled = PrgHandleEvent(CBDataP->progressP, &event);

         if (!handled) {  // Did the user tap the "Cancel" button?
            if( PrgUserCancel(CBDataP->progressP) )
               return exgErrUserCancel;
         }
      } while(event.eType != sysEventNilEvent);
   }

   return errNone;
}

static Err ImportFile(UInt16 volRefNum, Char *pathP, Char *nameP,
   UInt16 *cardNoP, LocalID *dbIDP)
{
   CBDataType userData;
   Char       fullPathP[256];
   Err        err;

   userData.progressP = NULL;
   userData.nameP = nameP;

   StrPrintF(fullPathP, "%s/%s", pathP, nameP); // rebuild full path to the
file
```

```
   err = VFSImportDatabaseFromFileCustom(volRefNum, fullPathP, cardNoP, dbIDP,
      CopyProgressCB, &userData);

   if (userData.progressP) // If the progress dialog was displayed, remove it.
      PrgStopDialog(userData.progressP, (err == exgErrUserCancel) );

   return err;
}
```

Exploring Palm OS Databases on Expansion Cards

The VFS Manager includes functions specifically designed for exploring the contents of a Palm OS database located on an expansion card. This access is read-only, however. You can extract individual records and resources from a database, and you can determine information such as the last modification date of a database on an expansion card. But there aren't parallel functions to write records and resources to a database or to update database-specific information for a database that is located on an expansion card. To do this you need to import the database into main memory, make the necessary changes, and then export it back to the expansion card.

To obtain a single record from a database located on an expansion card without first importing the database into main memory, use `VFSFileDBGetRecord()`. This function is analogous to `DmGetRecord()` but works with files on an external card rather than with databases in main memory. It transfers the specified record to the storage heap after allocating a handle of the appropriate size. Note that you'll need to free this memory, using `MemHandleFree()`, when the record is no longer needed.

The `VFSFileDBGetResource()` function operates in a similar fashion, but instead of loading a particular database record it loads a specified resource from a resource database located on an expansion card. This resource is put onto the storage heap. Again, free this memory once the resource is no longer needed.

To obtain more general information about a database on an external card, use `VFSFileDBInfo()`. In addition to the information you could obtain about any file on an external card using the

`VFSFileGetAttributes()` and `VFSFileGetDate()` functions, `VFSFileDBInfo()` returns:

- the database name
- the version of the database
- the number of times the database was modified
- the application info block handle
- the sort info block handle
- the database's type
- the database's creator
- the number of records in the database

NOTE: The functions described in this section incur a lot of overhead in order to parse the database file format. Frequent use of these functions is not recommended. Also, if you request either the application info block handle or the sort info block handle, you must free the handle when it is no longer needed.

Directory Operations

Many of the familiar operations you'd use to operate on directories are supported by the VFS Manager; these are listed in "Common Operations", below. One common operation—determining the files that are contained within a given directory—is covered in some detail in "Enumerating the Files in a Directory" on page 88. To improve data interchange with devices that aren't running Palm OS, expansion card manufacturers have specified default directories for certain file types. "Determining the Default Directory for a Particular File Type" on page 89 discusses how you can both determine and set the default directory for a given file type.

Directory Paths

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

Common Operations

The VFS Manager provides many of the standard directory operations that should be familiar from desktop and larger computer systems. Because these functions work largely as you would expect, their use isn't detailed here. See the descriptions of each individual function in Chapter 8, "VFS Manager," for the arguments, return values, and side effects of each.

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

Table 3.2 Common directory operations

Function	Description
`VFSDirCreate()`	Create a new directory.
`VFSFileDelete()`	Delete a directory, given a path.
`VFSFileRename()`	Rename a directory.
`VFSFileOpen()`	Open the file or directory.
`VFSFileClose()`	Close the file or directory.
`VFSFileGetAttributes()`	Obtain the attributes of an open directory, including hidden, read-only, system, and archive bits. See "File and Directory Attributes" in Chapter 8, "VFS Manager," for the bits that make up the attributes field.
`VFSFileSetAttributes()`	Set the attributes of an open directory, including hidden, read-only, system, and archive bits.
`VFSFileGetDate()`	Get the created, modified, and last accessed dates for an open file.
`VFSFileSetDate()`	Set the created, modified, and last accessed dates for an open file.

Enumerating the Files in a Directory

Enumerating the files within a directory is made simple due to the presence of the `VFSDirEntryEnumerate()` function. The use of this function is illustrated below. Note that `volRefNum` and `dirPathStr` must be declared and initialized prior to the following code.

Listing 3.2 Enumerating a directory's contents

```
// Open the directory and iterate through the files in it.
// volRefNum must have already been defined.
err = VFSFileOpen(volRefNum, "/", vfsModeRead, &dirRef);
if(err == errNone) {
   // Iterate through all the files in the open directory
   UInt32 fileIterator;
   FileInfoType fileInfo;
   FileRef dirRef;
   Char *fileName = MemPtrNew(256);   // should check for err

   fileInfo.nameP = fileName;    // point to local buffer
   fileInfo.nameBufLen = 256;
   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
   }
   MemPtrFree(fileName);
}
```

Each time through the while loop, `VFSDirEntryEnumerate()` sets the `FileInfoType` structure as appropriate for the file currently being enumerated. Note that if you want the filename, it isn't enough to simply allocate space for the `FileInfoType` structure; you must also allocate a buffer for the filename, set the appropriate pointer to it in the `FileInfoType` structure, and specify your buffer's length. Since the only other information encapsulated within `FileInfoType` is the file's attributes, most applications will want to also know the file's name.

Note that enumeration in the VFS Manager assumes that you are not changing the file set being enumerated. That is, you cannot delete or add files without restarting the enumeration.

Determining the Default Directory for a Particular File Type

As explained in "Standard Directories" on page 71, the expansion capabilities of Palm OS include a mechanism to map MIME types or file extensions to specific directory names. This mechanism is specific to the block device driver: where an image might be stored in the "/Images" directory on a Memory Stick, on an MMC card it may be stored in the "/DCIM" directory. The VFS Manager includes a function that enables you to get the default directory on a particular volume for a given file extension or MIME type, along with functions that allow you to register and un-register your own default directories.

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

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

`VFSGetDefaultDirectory()` allows you to determine the directory associated with a particular file suffix. However, there's no way to get the entire list of file suffixes that are mapped to default directories. For this reason, CardInfo keeps its own list of possible file suffixes. It iterates through this list, calling `VFSGetDefaultDirectory()` for each file suffix to get the full path to the corresponding default directory. It then looks into each default directory for files that match the expected suffix or suffixes for that directory.

Registering New Default Directories

In addition to the default directories that the underlying driver is already aware of, you can create your own mappings between files of a given type and a specific directory on a particular kind of external storage card. Most applications don't need this functionality; it is generally used by a block device driver to register those files and media types that are supported by that driver. However, `VFSRegisterDefaultDirectory()` and its opposite, `VFSUnregisterDefaultDirectory()`, are available to those applications that need them. Such applications should generally register the desired file types for `expMediaType_Any`. This is a wildcard which works for all media types; it can be overridden by a registration that specifies a real media type.

NOTE: Registering a directory as the default location for files of a given type on a particular type of media doesn't automatically register that file type with HotSync Exchange. See "HotSync Exchange" on page 138 of *Exploring Palm OS: High-Level Communications* for information on registering file types with HotSync Exchange.

If a default directory has already been registered for a given file/media type combination, applications should use the pre-existing registration instead of establishing a new one. Existing registrations should generally not be removed.

Default Directories Registered at Initialization

The VFS Manager registers the following under the `expMediaType_Any` media type, which `VFSGetDefaultDirectory()` reverts to when there is no default registered by the block device driver for a given media type.

Table 3.3 Default registrations

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

These registrations are intended to aid applications developers, but you aren't required to follow them. Although you can choose to ignore these registrations, by following them you'll improve interoperability between applications and other devices. For example, a digital camera which conforms to the media specifications will put its pictures into the registered directory (or a subdirectory of it) appropriate for the image format and media type. By looking up the registered directory for that format, an image viewer application on the handheld can easily find the images without having to search the entire card. These registrations also help prevent different developers from hard-coding different paths for specific file types. Thus, if a user has two different image viewer applications, both will look in the same location and find the same set of images.

Registering these file types at initialization allows you to use the HotSync® process to transfer files of these types to an expansion card. During the HotSync process, files of the registered types are placed directly in the specified directories on the card.

Custom Calls

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

Parameters are passed to the file system's custom function through a single `VFSCustomControl()` parameter. This parameter, *`valueP`*, is declared as a `void *` so you can pass a pointer to a structure of any type. A second parameter, *`valueLenP`*, allows you to specify the length of *`valueP`*. Note that these values are simply passed to the file system and are in reality dependent upon the underlying file system. See the description of `VFSCustomControl()` in Chapter 8, "VFS Manager," for more information.

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

Custom I/O

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

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

Summary of VFS Manager

VFS Manager Functions

Working with Files

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

Working with Directories

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

Working with Volumes

VFSVolumeEnumerate()
VFSVolumeFormat()
VFSVolumeGetLabel()
VFSVolumeInfo()
VFSVolumeMount()
VFSVolumeSetLabel()
VFSVolumeSize()
VFSVolumeUnmount()

Miscellaneous Functions

VFSCustomControl()
VFSExportDatabaseToFile()
VFSExportDatabaseToFileCustom()
VFSFileDBInfo()
VFSFileDBGetRecord()
VFSFileDBGetResource()
VFSImportDatabaseFromFile()
VFSImportDatabaseFromFileCustom()

VFS Manager Functions	
Compatibility Functions	
`VFSExportDatabaseToFileCustomV40()`	`VFSImportDatabaseFromFileCustomV40()`
`VFSExportDatabaseToFileV40()`	`VFSImportDatabaseFromFileV40()`

Part II
Reference

This part contains reference documentation for the following:

4

Data Manager

This chapter describes the Data Manager APIs. These APIs are those structures, constants, and functions that operate on extended and classic databases (collectively, the “non-schema” databases). This chapter is organized as follows:

The header file `DataMgr.h` declares the API that this chapter describes.

For more information on Palm OS® databases, see Chapter 2, “Palm OS Databases,” on page 11.

Data Manager Structures and Types

CategoryID Typedef

Purpose Container for a category's unique identifier.

Declared In `DataMgr.h`

Prototype
```
typedef int32_t CategoryID
```

DatabaseID Typedef

Purpose Container for a database's unique identifier.

Declared In `DataMgr.h`

Prototype
```
typedef uint32_t DatabaseID
```

DmBackupRestoreStateType Struct

Purpose Opaque container for the backup state, used to maintain state across multiple calls to `DmBackupUpdate()` or `DmRestoreUpdate()`.

Declared In `DataMgr.h`

Prototype
```
typedef struct DmBackupRestoreStateTag {
   uint32_t info[12];
} DmBackupRestoreStateType
typedef DmBackupRestoreStateType
*DmBackupRestoreStatePtr
```

Fields `info`

The backup state.

Comments Your application allocates a structure of this type and passes it to `DmBackupInitialize()` (or `DmRestoreInitialize()`) for initialization prior to serializing a database (or restoring a database that has been serialized). After passing it to `DmBackupUpdate()` (`DmRestoreUpdate()`), calling that function as many times as necessary, your application must pass it to `DmBackupFinalize()` (`DmRestoreFinalize()`) before releasing the storage occupied by the structure.

NOTE: The contents of this structure are opaque; your application should not attempt to directly manipulate the contents of this structure in any way.

DmDatabaseInfoType Struct

Purpose Data structure used to return information about a database through a call to `DmDatabaseInfo()`.

Declared In `DataMgr.h`

Prototype

```
typedef struct DmDatabaseInfoTag {
   uint32_t size;
   char *pName;
   char *pDispName;
   uint16_t *pAttributes;
   uint16_t *pVersion;
   uint32_t *pType;
   uint32_t *pCreator;
   uint32_t *pCrDate;
   uint32_t *pModDate;
   uint32_t *pBckpDate;
   uint32_t *pModNum;
   MemHandle *pAppInfoHandle;
   MemHandle *pSortInfoHandle;
   uint16_t *pEncoding;
} DmDatabaseInfoType
typedef DmDatabaseInfoType *DmDatabaseInfoPtr
```

Fields

`size`
Size of this structure.

`pName`
The database's name. This should be a pointer to 32-byte character array for this parameter, or `NULL` if you don't care about the name.

`pDispName`
(Schema databases only) The database's display name.

`pAttributes`
: The database's attribute flags. The section "Database Attributes" lists constants you can use to query the values returned in this parameter.

`pVersion`
: The application-specific version number. The default version number is 0.

`pType`
: The database's type, specified when it is created.

`pCreator`
: The database's creator, specified when it is created.

`pCrDate`
: The date the database was created, expressed as the number of seconds since the start of the Unix epoch.

`pModDate`
: The date the database was last modified, expressed as the number of seconds since the start of the Unix epoch.

`pBckpDate`
: The date the database was backed up, expressed as the number of seconds since the start of the Unix epoch.

`pModNum`
: The modification number, which is incremented every time a record in the database is added, modified, or deleted.

`pAppInfoHandle`
: *(Non-schema databases only)* Handle of the application info block, or `NULL`. The application info block is an optional field that the database may use to store application-specific information about the database.

`pSortInfoHandle`
: *(Non-schema databases only)* Handle of the database's sort table. This is an optional field in the database header.

`pEncoding`
: *(Schema databases only)* The database's encoding.

Comments Prior to calling `DmDatabaseInfo()`, initialize the fields of this structure to point to variables where `DmDatabaseInfo()` will write the information. If you don't want to retrieve data corresponding to a given field, set that field to `NULL`. See the comments section for `DmGetNextDatabaseByTypeCreator()` for an example of how this structure is initialized and used.

The fields representing dates (`pCrDate`, `pModDate`, `pBckpDate`) contain the number of non-leap seconds since the start of the Unix epoch: 00:00:00 UTC on Jan 1, 1970. Note that this is different from the way dates are returned by PACE, and is different from the way they are returned by `DmDatabaseInfoV50()`; PACE and `DmDatabaseInfoV50()` return dates based upon the "Palm OS epoch": the number of seconds since the beginning of Jan 1, 1904, local time.

DmFindType Typedef

Purpose Flags that indicate the type of database to be searched for when using `DmFindDatabase()`, `DmFindDatabaseByTypeCreator()`, or `DmOpenIteratorByTypeCreator()`. These flags can be OR'd together to search for a combination of database types.

Declared In `DataMgr.h`

Prototype `typedef uint32_t DmFindType`

Constants

`#define dmFindClassicDB ((DmFindType)0x00000004)`
Classic databases.

`#define dmFindExtendedDB ((DmFindType)0x00000002)`
Extended databases.

`#define dmFindSchemaDB ((DmFindType)0x00000001)`
Schema databases.

```
#define dmFindAllDB (dmFindSchemaDB |
  dmFindExtendedDB | dmFindClassicDB)
```
A convenience value that can be used when searching for databases of any type.

See Also Chapter 2, "Palm OS Databases," on page 11

DmOpenModeType Typedef

Purpose Type that holds the mode in which a database can be opened. You pass one or more of the associated constants as a parameter to

DmOpenDatabase(), DmOpenDatabaseByTypeCreator(), or DmOpenDBNoOverlay(). These constants are also used when working with schema databases using either DbOpenDatabase() or DbOpenDatabaseByName().

Declared In `DataMgr.h`

Prototype `typedef uint16_t DmOpenModeType;`

Constants

`#define dmModeExclusive ((DmOpenModeType)0x0008)`
While the database is open don't let anyone else open it. This value cannot be passed to DbOpenDatabase() and DbOpenDatabaseByName().

`#define dmModeReadOnly ((DmOpenModeType)0x0001)`
Open the database with read-only access. This value can be passed to DbOpenDatabase() and DbOpenDatabaseByName().

`#define dmModeReadWrite ((DmOpenModeType)0x0003)`
Open the database with read-write access. This value can be passed to DbOpenDatabase() and DbOpenDatabaseByName(). Use `dmModeWrite` when calling any of the `DmOpen...` functions.

`#define dmModeShowSecret ((DmOpenModeType)0x0010)`
Show records marked private. This value can be passed to DbOpenDatabase() and DbOpenDatabaseByName().

`#define dmModeWrite ((DmOpenModeType)0x0002)`
Open the database with write-only access. This value cannot be passed to DbOpenDatabase() and DbOpenDatabaseByName(); use `dmModeReadWrite` when calling one of these functions.

DmOpenRef Struct

Purpose Defines a pointer to an open database.

Declared In `DataMgr.h`

Prototype `typedef struct _opaque *DmOpenRef`

Fields None.

Comments The database pointer is created and returned by `DmOpenDatabase()`. It is used in any function that requires access to an open database.

DmResourceID Typedef

Purpose Defines a resource identifier. You assign each resource an ID at creation time.

Declared In `DataMgr.h`

Prototype `typedef uint16_t DmResourceID`

Comments Resource IDs greater than or equal to 10000 are reserved for system use.

DmResourceType Typedef

Purpose Defines the type of a resource.

Declared In `DataMgr.h`

Prototype `typedef uint32_t DmResourceType`

Comments The resource type is a four-character code such as `'Tbmp'` for bitmap resources.

DmSearchStateType Struct

Purpose Opaque container for the search state, used to maintain state when iterating through databases that match a specified type and creator.

Declared In `DataMgr.h`

Prototype

```
typedef struct {
   uint32_t info[8];
} DmSearchStateType
typedef DmSearchStateType *DmSearchStatePtr
```

Fields `info`

The search state.

Comments Your application should allocate a `DmSearchStateType` structure and pass it as the *`stateInfoP`* parameter when iterating through databases with `DmOpenIteratorByTypeCreator()`, `DmGetNextDatabaseByTypeCreator()`, and `DmCloseIteratorByTypeCreator()`; or when calling `DmGetNextDatabaseByTypeCreatorV50()`. These functions store private information in this structure and use that information if the search is continued.

NOTE: The contents of this structure are opaque; your application should not attempt to directly manipulate the contents of this structure in any way.

DmSortRecordInfoType Struct

Purpose Specifies information that can be used to sort a record.

Declared In `DataMgr.h`

Prototype

```
typedef struct {
   uint8_t attributes;
   uint8_t uniqueID[3];
} DmSortRecordInfoType
typedef DmSortRecordInfoType *DmSortRecordInfoPtr
```

Fields

`attributes`
The record's attributes. See "Non-Schema Database Record Attributes."

`uniqueID`
The unique identifier for the record.

Comments The database sorting functions (`DmInsertionSort()` and `DmQuickSort()`) pass this structure to your comparison callback function (of type `DmCompareFunctionType()`), where you can use the information therein to help when comparing two records. To create this structure, you can call `DmRecordInfoV50()`, which returns these values for a given record.

DmStorageInfoType Struct

Purpose Returns storage heap memory usage information through a call to `DmGetStorageInfo()`.

Declared In `DataMgr.h`

Prototype

```
typedef struct DmStorageInfoTag {
   uint32_t size;
   uint32_t bytesTotal;
   uint32_t bytesNonSecureUsed;
   uint32_t bytesNonSecureFree;
   uint32_t bytesSecureUsed;
   uint32_t bytesSecureFree;
   uint32_t bytesFreePool;
} DmStorageInfoType
typedef DmStorageInfoType *DmStorageInfoPtr
```

Fields

`size`
Size of this structure.

`bytesTotal`
Total amount of memory available for persistent storage.

`bytesNonSecureUsed`
Amount of memory used in non-secure storage.

`bytesNonSecureFree`
Amount of free memory in non-secure storage.

`bytesSecureUsed`
Amount of memory used in secure storage.

`bytesSecureFree`
Amount of free memory in secure storage.

`bytesFreePool`
Amount of memory in the free pool, available for both secure and non-secure storage.

Data Manager Constants

Non-Schema Database Record Attributes

Purpose These constants define the set of attributes that a non-schema database record can have. Use `DmGetRecordAttr()` to obtain a database record's attributes.

Declared In `DataMgr.h`

Constants

```
#define dmAllRecAttrs ( dmRecAttrDelete |
  dmRecAttrDirty | dmRecAttrBusy | dmRecAttrSecret
  )
```
The complete set of record attributes.

`#define dmRecAttrBusy 0x20`
The application has locked access to the record. A call to `DmGetRecord()` fails on a record that has this bit set.

`#define dmRecAttrDelete 0x80`
The record has been deleted.

`#define dmRecAttrDirty 0x40`
The record has been modified since the last sync.

`#define dmRecAttrSecret 0x10`
The record is private.

`#define dmSysOnlyRecAttrs ( dmRecAttrBusy )`
Mask that identifies those attributes that only the system can change.

`#define dmRecAttrCategoryMask ( (uint8_t) 0x0F )`
Mask that isolates the record's category.

Database Attributes

Purpose Define the set of attributes that a database can have. These attributes apply to schema, extended, and classic databases.

Declared In `DataMgr.h`

Constants

```
#define dmAllHdrAttrs (dmHdrAttrResDB |
  dmHdrAttrReadOnly | dmHdrAttrAppInfoDirty |
  dmHdrAttrBackup | dmHdrAttrOKToInstallNewer |
  dmHdrAttrResetAfterInstall |
  dmHdrAttrCopyPrevention | dmHdrAttrStream |
  dmHdrAttrHidden | dmHdrAttrLaunchableData |
  dmHdrAttrRecyclable | dmHdrAttrBundle |
  dmHdrAttrSchema | dmHdrAttrSecure |
  dmHdrAttrOpen)
```

A mask used to specify all header attributes.

`#define dmHdrAttrAppInfoDirty 0x0004`

The application info block is dirty (it has been modified since the last sync). This bit only applies to non-schema databases; schema databases don't have application info blocks.

`#define dmHdrAttrBackup 0x0008`

The database should be backed up to the desktop computer if no application-specific conduit is available.

`#define dmHdrAttrBundle 0x0800`

The database is bundled with its application during a beam, send, or copy operation. That is, if the user chooses to beam the application from the Launcher, the Launcher beams this database along with the application's resource database and overlay database. (Note that overlay databases are automatically beamed with the application database. You do not need to set this bit in overlay databases.)

`#define dmHdrAttrCopyPrevention 0x0040`

Prevents the database from being copied by methods such as IR beaming.

`#define dmHdrAttrHidden 0x0100`

This database should be hidden from view. For example, this attribute is set to hide some applications in the Launcher's main view. You can set it on record databases to have the Launcher disregard the database's records when showing a count of records.

`#define dmHdrAttrLaunchableData 0x0200`
: This database contains data but it can be "launched" from the Launcher.

`#define dmHdrAttrExtendedDB dmHdrAttrSecure`
: If `dmHdrAttrSchema` is not set, the database is an extended database. Note that this bit serves a dual-purpose, depending upon the `dmHdrAttrSchema` bit; if the database is a schema database (`dmHdrAttrSchema` is set), this bit indicates whether or not the schema database is a secure database. See Chapter 2, "Palm OS Databases," for an explanation of the differences between the various database types.

`#define dmHdrAttrOKToInstallNewer 0x0010`
: The backup conduit can install a newer version of this database with a different name if the current database is open. This mechanism is used to update the Graffiti 2 Shortcuts databases, for example.

`#define dmHdrAttrOpen 0x8000`
: The database is open.

`#define dmHdrAttrReadOnly 0x0002`
: The database is a read-only database.

`#define dmHdrAttrRecyclable 0x0400`
: The database is recyclable. Recyclable databases are deleted when they are closed or upon a system reset.

`#define dmHdrAttrResDB 0x0001`
: The database is a resource database.

`#define dmHdrAttrResetAfterInstall 0x0020`
: The device must be reset after this database is installed. That is, the HotSync® application forces a reset after installing this database.

`#define dmHdrAttrSchema 0x1000`
: The database is a schema database. See Chapter 2, "Palm OS Databases," for an explanation of the differences between the various database types.

`#define dmHdrAttrSecure 0x2000`
: The database is a secure database.

`#define dmHdrAttrStream 0x0080`
: The database is a file stream.

`#define dmSysOnlyHdrAttrs ( dmHdrAttrResDB | dmHdrAttrSchema | dmHdrAttrSecure | dmHdrAttrOpen )`
A mask specifying the attributes that only the system can change (open and resource database).

Miscellaneous Data Manager Constants

Purpose Miscellaneous constants defined by the Data Manager.

Declared In `DataMgr.h`

Constants

`#define appInfoStringsRsc 'tAIS'`
Application Info strings resource type.

`#define dmMaxRecordIndex ( (uint16_t) 0xFFFE )`
The highest record index that can be used with a classic database.

`#define dmAllCategories ( (uint8_t) 0xFF )`
Category value that can be supplied to `DmNumRecordsInCategory()` and `DmQueryNextInCategory()` to indicate all categories.

`#define dmCategoryLength 16`
The maximum length of a classic or extended database category name, in bytes, including the `NULL` terminator.

`#define dmDBNameLength 32`
The maximum length of a database name, in bytes, including the `NULL` terminator.

`#define dmDefaultRecordsID 0`
Records in a default database are copied with their unique ID seeds set to this value.

`#define dmInvalidRecIndex ( (uint16_t) -1 )`
Resource index value returned by `DmFindResource()` when that function fails to find the specified resource.

`#define dmRecNumCategories 16`
The maximum number of categories that can be used with a classic or extended database.

`#define dmRecordIDReservedRange 1`
: Upper limit of the range of unique ID seed values reserved for use by the operating system in conjunction with classic and extended databases.

`#define dmSearchWildcardID ((uint32_t)0)`
: A "wild card" that matches databases of any type and/or creator when iterating through databases with `DmOpenIteratorByTypeCreator()` or searching for databases with either `DmGetNextDatabaseByTypeCreator()` or `DmGetNextDatabaseByTypeCreatorV50()`.

`#define dmSeekBackward -1`
: Direction value supplied to `DmFindRecordByOffsetInCategory()` to indicate that the search should be performed from the specified position towards the beginning of the database.

`#define dmSeekForward 1`
: Direction value supplied to `DmFindRecordByOffsetInCategory()` to indicate that the search should be performed from the specified position towards the end of the database.

`#define dmUnfiledCategory 0`
: Category identifier for the Unfiled category.

`#define dmUnusedRecordID 0`
: A record ID value representing an illegal or unused record. A "real" record cannot use this value as its record identifier.

Data Manager Error Codes

Purpose Error codes returned by the various Data Manager functions. These codes are returned by schema database functions as well as classic database functions.

Declared In `DataMgr.h`

Constants `#define dmErrAccessDenied (dmErrorClass | 37)`
: The database is a secure database and you don't have permission to edit it.

`#define dmErrAlreadyExists (dmErrorClass | 25)`
Another database with the same name already exists.

`#define dmErrAlreadyOpenForWrites (dmErrorClass | 22)`
The database is already open with write access.

`#define dmErrBadOverlayDBName (dmErrorClass | 32)`
The length of the locale description or overlay database name is incorrect, or the locale description begins with an underscore ('_') character.

`#define dmErrBaseRequiresOverlay (dmErrorClass | 33)`
The base probably requires an overlay, but the corresponding overlay cannot be located.

`#define dmErrBufferNotLargeEnough (dmErrorClass | 42)`
While copying a table column value from a schema database, it was determined that the supplied buffer wasn't large enough to contain the column value.

`#define dmErrBuiltInProperty (dmErrorClass | 58)`
The schema database column property you are trying to alter is a built-in property; it cannot be changed or removed.

`#define dmErrCantFind (dmErrorClass | 7)`
The specified database can't be found.

`#define dmErrCantOpen (dmErrorClass | 6)`
The database cannot be opened.

`#define dmErrCategoryLimitReached (dmErrorClass | 74)`
The schema database row cannot be made a member of the specified category because it is already a member of the maximum number of allowable categories.

`#define dmErrColumnDefinitionsLocked (dmErrorClass | 76)`
The schema database table's column definitions are locked.

`#define dmErrColumnIDAlreadyExists (dmErrorClass | 46)`
The specified schema database table already contains a column with the specified ID.

`#define dmErrColumnIndexOutOfRange (dmErrorClass | 43)`
: The supplied column index exceeds the number of columns in the schema database table.

`#define dmErrColumnNameAlreadyExists (dmErrorClass | 70)`
: The specified schema database table already contains a column with the specified name.

`#define dmErrColumnPropertiesLocked (dmErrorClass | 75)`
: The specified column property is locked.

`#define dmErrCorruptDatabase (dmErrorClass | 9)`
: The database is corrupted.

`#define dmErrDatabaseNotProtected (dmErrorClass | 28)`
: `DmDatabaseProtectV50()` failed to protect the specified database.

`#define dmErrDatabaseOpen (dmErrorClass | 5)`
: The function cannot be performed on an open database, and the database is open.

`#define dmErrDatabaseProtected (dmErrorClass | 27)`
: The database is marked as protected.

`#define dmErrDeviceLocked (dmErrorClass | 59)`

`#define dmErrEncryptionFailure (dmErrorClass | 54)`

`#define dmErrIndexOutOfRange (dmErrorClass | 2)`
: The specified index is out of range.

`#define dmErrInvalidCategory (dmErrorClass | 18)`
: At least one of the supplied category IDs is not a valid schema database category.

`#define dmErrInvalidColSpec (dmErrorClass | 40)`
: At least one of the specified schema database table column attributes is not a valid column attribute.

`#define dmErrInvalidColType (dmErrorClass | 41)`
: The specified schema database table column type is not a valid column type.

`#define dmErrInvalidColumnID (dmErrorClass | 44)`
: One or more of the specified column IDs doesn't correspond to a column in the specified schema database table.

`#define dmErrInvalidColumnName (dmErrorClass | 79)`
: The supplied column name doesn't correspond to a column within the schema database table.

`#define dmErrInvalidDatabaseName (dmErrorClass | 26)`
: The name you've specified for the database is invalid.

`#define dmErrInvalidID (dmErrorClass | 30)`
: The schema database row ID is invalid.

`#define dmErrInvalidIndex (dmErrorClass | 29)`
: The row or sort index value exceeds the number of rows or sort indices defined for the schema database table.

`#define dmErrInvalidTableName (dmErrorClass | 78)`
: The supplied table name doesn't correspond to a table in the schema database.

`#define dmErrInvalidOperation (dmErrorClass | 60)`
: The requested schema database operation is not valid.

`#define dmErrInvalidParam (dmErrorClass | 3)`
: The function received an invalid parameter.

`#define dmErrInvalidPrimaryKey (dmErrorClass | 66)`
: Not currently used.

`#define dmErrInvalidPropID (dmErrorClass | 56)`
: The specified schema database table column doesn't have a property with the specified property ID.

`#define dmErrInvalidSchemaDefn (dmErrorClass | 38)`
: You are creating a schema database or adding a table to an existing schema database and the supplied `DbTableDefinitionType` structure defining the new table is invalid.

`#define dmErrInvalidSizeSpec (dmErrorClass | 51)`
: You are creating a schema database or adding a table to an existing schema database and one of the table's vector column sizes is zero.

`#define dmErrInvalidSortDefn (dmErrorClass | 71)`
: You are adding a sort index to a schema database that is incorrectly specified or you are attempting to remove a sort index that isn't defined for the database.

`#define dmErrInvalidSortIndex (dmErrorClass | 65)`
: You are opening a schema database cursor and one of the specified sort IDs isn't defined for the specified database table.

`#define dmErrInvalidVectorType (dmErrorClass | 50)`
: You adding a vector column to an existing schema database—either explicitly or during the creation of a new schema database—but the specified column type isn't appropriate for a vector column.

`#define dmErrMemError (dmErrorClass | 1)`
: A memory error occurred.

`#define dmErrNoColumnData (dmErrorClass | 48)`
: Your request for the value of one or more schema database table columns cannot be fulfilled because the column contains no data.

`#define dmErrNoCustomProperties (dmErrorClass | 57)`
: The schema database contains no custom properties.

`#define dmErrNoData (dmErrorClass | 53)`
: The specified schema database table has no columns defined.

`#define dmErrNoMoreData (dmErrorClass | 72)`
: The backup operation is complete. See `DmBackupUpdate()` for a detailed explanation and example of how this error code is used.

`#define dmErrNoOpenDatabase (dmErrorClass | 17)`
: The function is to search all open databases, but there are none.

`#define dmErrNotRecordDB (dmErrorClass | 12)`
: You've attempted to perform a record function on a resource database.

`#define dmErrNotResourceDB (dmErrorClass | 13)`
You've attempted to perform a Resource Manager operation on a record database.

`#define dmErrNotSchemaDatabase (dmErrorClass | 35)`
The specified database is not a schema database.

`#define dmErrNotSecureDatabase (dmErrorClass | 36)`
The specified database is not a secure schema database.

`#define dmErrNotValidRecord (dmErrorClass | 19)`
The record handle is invalid.

`#define dmErrNoUserPassword (dmErrorClass | 68)`
The Authorization Manager doesn't have a user password on file.

`#define dmErrOneOrMoreFailed (dmErrorClass | 62)`
At least one of the schema database table's column definitions could not be retrieved.

`#define dmErrOpenedByAnotherTask (dmErrorClass | 23)`
You've attempted to open a database that another task already has open.

`#define dmErrOperationAborted (dmErrorClass | 73)`
The variables bound to a schema database cursor couldn't be written to the database, or a database backup or restore operation was aborted.

`#define dmErrReadOnly (dmErrorClass | 4)`
You've attempted to write to or modify a database that is open in read-only mode.

`#define dmErrReadOutOfBounds (dmErrorClass | 49)`
A schema database table vector column is being read in which the specified offset exceeds the bounds of the column.

`#define dmErrRecordArchived (dmErrorClass | 11)`
The function requires that the record not be archived, but it is.

`#define dmErrRecordBusy (dmErrorClass | 15)`
The function requires that the record not be busy, but it is.

`#define dmErrRecordDeleted (dmErrorClass | 10)`
The record has been deleted.

`#define dmErrRecordInWrongCard (dmErrorClass | 8)`
You've attempted to attach a record to a database when the record and database reside on different memory cards.

`#define dmErrTableNotEmpty (dmErrorClass | 61)`
An attempt to remove a schema database table failed because the table isn't empty.

`#define dmErrResourceNotFound (dmErrorClass | 16)`
The resource can't be found.

`#define dmErrROMBased (dmErrorClass | 14)`
You've attempted to delete or modify a ROM-based database.

`#define dmErrSchemaBase (dmErrorClass | 34)`
Not an actual error code: this value serves to mark the beginning of the set of error codes created specifically for schema databases.

`#define dmErrSchemaIndexOutOfRange (dmErrorClass | 47)`
The supplied table index exceeds the number of tables in the schema database.

`#define dmErrTableNameAlreadyExists (dmErrorClass | 69)`
The schema database to which you are attempting to add a new table already contains a table with the supplied name, or, during the creation of a new schema database, you specified the same table name more than once.

`#define dmErrSchemaNotFound (dmErrorClass | 55)`
Not currently used.

`#define dmErrSeekFailed (dmErrorClass | 21)`
The operation of seeking the next record in the category failed.

`#define dmErrSortDisabled (dmErrorClass | 67)`
Not currently used.

`#define dmErrSQLParseError (dmErrorClass | 78)`
The SQL used to specify the schema database sort index is incorrectly formatted.

`#define dmErrUniqueIDNotFound (dmErrorClass | 24)`
A record with the specified unique ID can't be found.

`#define dmErrUnknownLocale (dmErrorClass | 31)`
The specified locale is unknown to the operating system.

`#define dmErrCursorBOF (dmErrorClass | 63)`
The schema database cursor position—either the current position or the one specified—is located before the first row in the cursor.

`#define dmErrCursorEOF (dmErrorClass | 64)`
The schema database cursor position—either the current position or the one specified—is located after the last row in the cursor.

`#define dmErrWriteOutOfBounds (dmErrorClass | 20)`
A write operation exceeded the bounds of the record.

Data Manager Functions and Macros

DmArchiveRecord Function

Purpose Mark a record as archived by leaving the record's chunk intact and setting the delete bit for the next HotSync operation.

Declared In `DataMgr.h`

Prototype
```
status_t DmArchiveRecord (DmOpenRef dbRef,
   uint16_t index)
```

Parameters

→ *`dbRef`*
`DmOpenRef` to an open database.

→ *`index`*
Which record to archive.

Returns Returns `errNone` if no error, or one of the following if an error occurs:

`dmErrReadOnly`
You've attempted to write to or modify a database that is open in read-only mode.

`dmErrIndexOutOfRange`
The specified index is out of range.

`dmErrRecordArchived`
: The function requires that the record not be archived, but it is.

`dmErrRecordDeleted`
: The record has been deleted.

`memErrInvalidParam`
: A memory error occurred.

Some releases may display a fatal error message instead of returning the error code.

Comments When a record is archived, the deleted bit is set but the chunk is not freed and the record ID is preserved. This way, the next time the user synchronizes with the desktop system, the conduit can save the record data on the desktop before it permanently removes the record entry and data from the Palm Powered™ device.

Based on the assumption that a call to `DmArchiveRecord()` indicates that you are finished with the record and aren't going to refer to it again, this function sets the chunk's lock count to zero.

See Also DmRemoveRecord(), DmDetachRecord(), DmNewRecord(), DmDeleteRecord()

DmAttachRecord Function

Purpose Attach an existing chunk ID handle to a database as a record.

Declared In `DataMgr.h`

Prototype

```
status_t DmAttachRecord (DmOpenRef dbRef,
   uint16_t *pIndex, MemHandle hNewRecord,
   MemHandle *hReplacedRecord)
```

Parameters

→ *`dbRef`*
: `DmOpenRef` to an open database.

↔ *`pIndex`*
: Pointer to the index where the new record should be placed. Specify the value `dmMaxRecordIndex` to add the record to the end of the database.

→ *`hNewRecord`*
: Handle of the new record.

↔ *`hReplacedRecord`*
: If non-`NULL` upon entry, indicates that the record at `*`*`pIndex`* should be replaced. Upon return, contains the handle to the replaced record.

Returns Returns `errNone` if no error, or one of the following if an error occurs:

`dmErrMemError`
: A memory error occurred.

`memErrChunkLocked`
: The associated memory chunk is locked.

`memErrInvalidParam`
: A memory error occurred.

`memErrNotEnoughSpace`
: A memory error occurred.

`dmErrReadOnly`
: You've attempted to write to or modify a database that is open in read-only mode.

`dmErrNotRecordDB`
: You've attempted to perform a record function on a resource database.

`dmErrRecordInWrongCard`
: You've attempted to attach a record to a database when the record and database reside on different memory cards.

`dmErrIndexOutOfRange`
: The specified index is out of range.

Some releases may display a fatal error message instead of returning some of these error codes.

Comments Given the handle of an existing chunk, this function makes that chunk a new record in a database and sets the dirty bit. The parameter *`pIndex`* points to an index variable. If *`hReplacedRecord`* is `NULL`, the new record is inserted at index **`pIndex`* and all record indices that follow are shifted down. If **`pIndex`* is greater than the number of records currently in the database, the new record *`hNewRecord`* is appended to the end and its index is returned in **`pIndex`*. If *`hReplacedRecord`* is not `NULL`, the new record replaces an existing record at index **`pIndex`* and the handle of the old record is returned in **hReplacedRecord* so that the application can free it or attach it to another database.

This function is useful for cutting and pasting between databases.

See Also DmRemoveRecord(), DmDetachRecord(), DmNewRecord(), DmDeleteRecord()

DmAttachResource Function

Purpose Attach an existing chunk ID to a resource database as a new resource.

Declared In `DataMgr.h`

Prototype

```
status_t DmAttachResource (DmOpenRef dbRef,
   MemHandle hNewRes, DmResourceType resType,
   DmResourceID resID)
```

Parameters

→ *dbRef*
  `DmOpenRef` to an open database.

→ *hNewRes*
  Handle of new resource's data.

→ *resType*
  Type of the new resource.

→ *resID*
  ID of the new resource.

Returns Returns `errNone` if no error, or one of the following if an error occurs:

`dmErrMemError`
  A memory error occurred.

`memErrChunkLocked`
  The associated memory chunk is locked.

`memErrInvalidParam`
  A memory error occurred.

`memErrNotEnoughSpace`
  A memory error occurred.

`dmErrReadOnly`
: You've attempted to write to or modify a database that is open in read-only mode.

`dmErrRecordInWrongCard`
: You've attempted to attach a record to a database when the record and database reside on different memory cards.

Some releases may display a fatal error message instead of returning some of these error codes. All releases may display a fatal error message if the database is not a resource database.

Comments Given the handle of an existing chunk with resource data in it, this function makes that chunk a new resource in a resource database. The new resource will have the given type and ID.

See Also DmDetachResource(), DmRemoveResource(), DmNewHandle(), DmNewResource()

DmBackupFinalize Function

Purpose Complete or abort an on-going database backup operation.

Declared In `DataMgr.h`

Prototype

```
status_t DmBackupFinalize
    (DmBackupRestoreStatePtr pState,
    Boolean fAbort)
```

Parameters

→ *`pState`*
: Pointer to a `DmBackupRestoreStateType` structure allocated by the caller and initialized with `DmBackupInitialize()`.

→ *`fAbort`*
: Set to `true` to abort an on-going backup operation, or `false` to clean up after a successful backup.

Returns Returns `errNone` if the database image was successfully created, `dmErrOperationAborted` if the backup operation was cancelled, or one of the following errors otherwise:

`dmErrInvalidParam`
: One of the parameters is invalid or corrupt.

`dmErrMemError`
: A memory error occurred which prevented the backup operation from completing.

Comments This function allows the Data Manager to perform a final clean up of the internal structures it allocated for the operation. Applications should always call this function after having started a backup operation, whether or not the backup completed successfully. See `DmBackupUpdate()` for sample code illustrating this function's use.

The backup operation can be used with schema, extended, or classic databases.

See Also `DmBackupInitialize()`, `DmRestoreFinalize()`

DmBackupInitialize Function

Purpose Initialize the Data Manager prior to starting a backup operation on the specified database.

Declared In `DataMgr.h`

Prototype

```
status_t DmBackupInitialize
    (DmBackupRestoreStatePtr pState,
    DatabaseID dbID)
```

Parameters

↔ *`pState`*
: Pointer to a `DmBackupRestoreStateType` structure allocated by the caller.

→ *`dbID`*
: Database ID of the database to be backed up.

Returns Returns `errNone` if the structure was successfully initialized, or one of the following if an error occurred:

`dmErrCantFind`
: The specified database doesn't exist.

`dmErrDatabaseOpen`
: The function cannot be performed on an open database, and the database is open.

`dmErrAccessDenied`
: The caller was not authorized to perform a backup operation for the specified database. This can be returned if the specified database is a secure schema database.

`dmErrInvalidParam`
: One of the parameters is invalid.

`dmErrMemError`
: A memory error occurred.

Comments Use `DmBackupInitialize()` to start a database backup operation. See `DmBackupUpdate()` for sample code illustrating this function's use.

The backup operation can be used with schema, extended, or classic databases.

IMPORTANT: When called from the main application thread, this function may block. While blocked, the application will not receive events and won't redraw its windows. As well, deferred sublaunches and notifications won't execute while the main application thread is blocked.

See Also `DmBackupFinalize()`, `DmRestoreInitialize()`

DmBackupUpdate Function

Purpose Stream a database into its corresponding image within the specified buffer.

Declared In `DataMgr.h`

Prototype

```
status_t DmBackupUpdate
    (DmBackupRestoreStatePtr pState,
    MemPtr pBuffer, uint32_t *pSize)
```

Parameters

→ *`pState`*
: Pointer to a `DmBackupRestoreStateType` structure allocated by the caller and initialized with `DmBackupInitialize()`.

→ *`pBuffer`*
: Pointer to a buffer to hold the backed-up database image.

↔ *pSize*
Before calling, set this variable to the size of the *pBuffer* data buffer. Upon return, it contains the actual number of bytes written to *pBuffer*.

Returns Returns `errNone` if the operation was successful, `dmErrNoMoreData` if the backup operation is complete, or one of the following if an error occurred:

`dmErrInvalidParam`
One of the parameters is invalid or corrupt.

`dmErrMemError`
A memory error occurred which prevented the backup operation from completing.

Comments Use `DmBackupUpdate()`, along with `DmBackupInitialize()` and `DmBackupFinalize()`, to get the serial image of a database.

You may need to call `DmBackupUpdate()` several times in order to get the complete image of the specified database. Call `DmBackupUpdate()` as many times as required and as long as it returns `errNone`, until it finally returns `dmErrNoMoreData`.

When `DmBackupUpdate()` returns an error code other than `errNone` or `dmErrNoMoreData`, the operation has been aborted due to a fatal error. You must still call `DmBackupFinalize()` in order to let the Data Manager perform its final clean up of the internal structures it allocated for the operation.

The backup operation can be used with schema, extended, or classic databases.

Example The following code shows how to use the `DmBackup...()` functions to send an image of a database to a fictitious serial channel.

```
status_t error;
DmBackupRestoreStateType backupState;
char buffer[BUFFER_SIZE];
uint32_t size;
Boolean fAbort;
Boolean fDone;

error = DmBackupInitialize(&backupState, dbID);

if (error == errNone){
   do {
      // Reset the size value with the buffer size for each
      // loop as this variable gets updated with the actual
      // number of bytes written to the buffer after each
      // call to DmBackupDatabase.
      size = sizeof(buffer);

      error = DmBackupUpdate(&backupState, &buffer, &size);

      fDone = (error == dmErrNoMoreData);

      if ((error == errNone) || fDone){
         // Stream the database image data chunk we got back
         // out to some I/O channel...
         error = SendDatabaseImageData(&buffer, size);
      }

      // Abort the operation if we got back an error or if
      // the user decided to cancel the operation...
      fAbort = (error != errNone) || DidUserCancel();

   } while(!fDone && !fAbort);


   // Always call DmBackupFinalize to complete the backup
   // operation, whether or not it completed successfully
   error = DmBackupFinalize(&backupState, fAbort);
}

if (error == errNone){
   // The backup operation completed successfully...
} else {
   if (error == dmErrOperationAborted){
      // The user aborted the operation
   } else {
      // Some other fatal error occurred...
   }
}
```

See Also DmRestoreUpdate()

DmCloseDatabase Function

Purpose Close a database.

Declared In `DataMgr.h`

Prototype `status_t DmCloseDatabase (DmOpenRef *dbRef*)`

Parameters → *`dbRef`*
`DmOpenRef` to an open database.

Returns Returns `errNone` if no error, or `dmErrInvalidParam` if an error occurs. Some releases may display a fatal error message instead of returning the error code.

Comments This function doesn't unlock any records that were left locked. Records and resources should not be left locked. If a record or resource is left locked, you should not use its reference because the record can disappear during a HotSync operation or if the database is deleted by the user. To prevent the database from being deleted, you can use `DmSetDatabaseProtection()` before closing.

If there is an overlay associated with the database passed in, `DmCloseDatabase()` closes the overlay as well.

If the database has the recyclable bit set (`dmHdrAttrRecyclable`), `DmCloseDatabase()` calls `DmDeleteDatabase()` to delete it.

`DmCloseDatabase()` updates the database's modification date.

See Also `DmOpenDatabase()`, `DmDeleteDatabase()`, `DmOpenDatabaseByTypeCreator()`

DmCloseIteratorByTypeCreator Function

Purpose Indicate that a particular iteration loop is complete.

Declared In `DataMgr.h`

Prototype
```
status_t DmCloseIteratorByTypeCreator
    (DmSearchStatePtr stateInfoP)
```

Parameters → *`stateInfoP`*
Pointer to the `DmSearchStateType` structure supplied to `DmOpenIteratorByTypeCreator()` and `DmGetNextDatabaseByTypeCreator()`.

Returns Returns `errNone`.

Comments See the comments under `DmGetNextDatabaseByTypeCreator()` for an example of how this function is used.

See Also `DmGetNextDatabaseByTypeCreator()`, `DmOpenIteratorByTypeCreator()`

DmCreateDatabase Function

Purpose Create a new extended database with the given name, creator, and type.

Declared In `DataMgr.h`

Prototype

```
status_t DmCreateDatabase (const char *nameP,
   uint32_t creator, uint32_t type,
   Boolean resDB)
```

Parameters

→ *`nameP`*
: Name of new database, up to 32 ASCII bytes long, including the null terminator (as specified by `dmDBNameLength`). Database names must use only 7-bit ASCII characters (0x20 through 0x7E).

→ *`creator`*
: Creator of the database.

→ *`type`*
: Type of the database.

→ *`resDB`*
: If `true`, create a resource database. If `false`, create a record database.

Returns Returns `errNone` if no error, or one of the following if an error occurs:

`dmErrInvalidDatabaseName`
: The name you've specified for the database is invalid.

`dmErrAlreadyExists`
: Another database with the same name already exists.

`memErrCardNotPresent`
: The specified card can't be found.

`dmErrMemError`
: A memory error occurred.

`memErrChunkLocked`
: The associated memory chunk is locked.

`memErrInvalidParam`
: A memory error occurred.

`memErrInvalidStoreHeader`
: The specified card has no storage RAM.

`memErrNotEnoughSpace`
: A memory error occurred.

`memErrRAMOnlyCard`
: The specified card has no storage RAM.

May display a fatal error message if the master database list cannot be found.

Comments If another database with the same name and creator already exists in RAM store, this function returns a `dmErrAlreadyExists` error.

Once created, the database ID can be retrieved by calling `DmFindDatabase()`. The database can be opened using the database ID.

After you create a database, you should call `DmSetDatabaseInfo()` to set the version number. Databases default to version 0 if the version isn't explicitly set.

IMPORTANT: This function creates extended databases only. To create a classic database, use `DmCreateDatabaseV50()`. To create a schema database, use `DbCreateDatabase()`.

See Also DmCreateDatabaseFromImage(), DmOpenDatabase(), DmDeleteDatabase()

DmCreateDatabaseFromImage Function

Purpose Create an entire database from a single resource that contains an image of the database.

Declared In `DataMgr.h`

Prototype
```
status_t DmCreateDatabaseFromImage
    (MemPtr pImage, DatabaseID *pDbID)
```

Parameters

→ *`pImage`*
Pointer to locked resource containing database image.

← *`pDbID`*
Pointer to a variable that will hold the ID of the newly-created database, or `NULL` if the ID isn't needed.

Returns Returns `errNone` if the operation completed successfully, or one of the following otherwise:

`dmErrInvalidParam`
`pImage` is `NULL`.

`dmErrMemError`
A memory error occurred. Most likely there wasn't enough memory available to create the database.

`dmErrCorruptDatabase`
The format of the database image is unrecognized.

`dmErrAlreadyExists`
The database being created already exists on the device.

Comments An image is the same as a desktop file representation of a PRC or PDB file. This function creates either an extended or a classic database, or a non-secure schema database, depending upon the image stored in the resource. To perform a similar operation for a secure schema database, see `DbCreateSecureDatabaseFromImage()`.

This function is intended for applications in the ROM to install default databases after a hard reset. RAM-based applications that want to install a default database should install a PDB file separately to save storage heap space.

See Also DmCreateDatabase(), DmOpenDatabase()

DmCreateDatabaseFromImageV50 Function

Purpose Create an entire classic database from a single resource that contains an image of the database.

Declared In `DataMgr.h`

Prototype
```
status_t DmCreateDatabaseFromImageV50
    (MemPtr pImage)
```

Parameters → *`pImage`*
Pointer to locked resource containing database image.

Returns Returns `errNone` if the operation completed successfully, or one of the following otherwise:

`dmErrInvalidParam`
`pImage` is `NULL`.

`dmErrMemError`
A memory error occurred. Most likely there wasn't enough memory available to create the database.

`dmErrCorruptDatabase`
The format of the database image is unrecognized.

`dmErrAlreadyExists`
The database being created already exists on the device.

Comments An image is the same as a desktop file representation of a PRC or PDB file.

This function is intended for applications in the ROM to install default databases after a hard reset. RAM-based applications that want to install a default database should install a PDB file separately to save storage heap space.

Compatibility This function is provided for compatibility purposes. Note that it works only with classic databases—the only type of database supported in PACE and by previous versions of Palm OS. Native Palm OS Cobalt applications will likely want to use `DmCreateDatabaseFromImage()` instead.

See Also `DmCreateDatabaseFromImage()`

DmCreateDatabaseV50 Function

Purpose Create a new classic database on the specified card with the given name, creator, and type.

Declared In `DataMgr.h`

Prototype

```
status_t DmCreateDatabaseV50 (uint16_t cardNo,
    const char *nameP, uint32_t creator,
    uint32_t type, Boolean resDB)
```

Parameters

→ *`cardNo`*
The number of the card on which to create the database. This value should always be zero.

→ *`nameP`*
Name of new database, up to 32 ASCII bytes long, including the null terminator (as specified by `dmDBNameLength`). Database names must use only 7-bit ASCII characters (0x20 through 0x7E).

→ *`creator`*
Creator of the database.

→ *`type`*
Type of the database.

→ *`resDB`*
If `true`, create a resource database.

Returns Returns `errNone` if no error, or one of the following if an error occurs:

`dmErrInvalidDatabaseName`
The name you've specified for the database is invalid.

`dmErrAlreadyExists`
Another database with the same name already exists.

`memErrCardNotPresent`
The specified card can't be found.

`dmErrMemError`
A memory error occurred.

`memErrChunkLocked`
The associated memory chunk is locked.

`memErrInvalidParam`
A memory error occurred.

`memErrInvalidStoreHeader`
The specified card has no storage RAM.

`memErrNotEnoughSpace`
A memory error occurred.

`memErrRAMOnlyCard`
The specified card has no storage RAM.

May display a fatal error message if the master database list cannot be found.

Comments Call this function to create a new database on a specific card. If another classic database with the same name already exists in RAM store, this function returns a `dmErrAlreadyExists` error code. Once created, the database ID can be retrieved by calling `DmFindDatabase()`. The database can be opened using the database ID. To create a resource database instead of a record-based database, set the *`resDB`* parameter to `true`.

After you create a database, it's recommended that you call `DmSetDatabaseInfo()` to set the version number. Databases default to version 0 if the version isn't explicitly set.

Compatibility This function is provided for compatibility purposes. Note that it only works with classic databases—the only type of database supported in PACE and by previous versions of Palm OS. Native Palm OS Cobalt applications may want to use `DmCreateDatabase()` instead.

See Also DmCreateDatabaseFromImage(), DmOpenDatabase(), DmDeleteDatabase()

DmDatabaseInfo Function

Purpose Retrieve information about a non-schema database.

Declared In `DataMgr.h`

Prototype
```
status_t DmDatabaseInfo (DatabaseID dbID,
   DmDatabaseInfoPtr pDatabaseInfo)
```

Parameters → *`dbID`*
Database ID of the database.

→ *pDatabaseInfo*
Pointer to a `DmDatabaseInfoType` structure that indicates where, or if, the database information is to be written.

Returns Returns `errNone` if the database information was successfully retrieved, or `dmErrInvalidParam` if an error occurred.

Comments Initialize the fields of the *`pDatabaseInfo`* structure to point to variables where this function will write the information. If you don't want to retrieve data corresponding to a given field, set that field to `NULL`.

See Also `DmDatabaseInfoV50()`, `DmSetDatabaseInfo()`, `DmDatabaseSize()`, `DmOpenDatabaseInfoV50()`, `DmFindDatabase()`, `DmGetNextDatabaseByTypeCreator()`, `TimSecondsToDateTime()`

DmDatabaseInfoV50 Function

Purpose Retrieve information about a database.

Declared In `DataMgr.h`

Prototype
```
status_t DmDatabaseInfoV50 (uint16_t cardNo,
    LocalID dbID, char *nameP,
    uint16_t *attributesP, uint16_t *versionP,
    uint32_t *crDateP, uint32_t *modDateP,
    uint32_t *bckUpDateP, uint32_t *modNumP,
    LocalID *appInfoIDP, LocalID *sortInfoIDP,
    uint32_t *typeP, uint32_t *creatorP)
```

Parameters
→ *cardNo*
Number of the card the database resides on.

→ *dbID*
Database ID of the database.

← *nameP*
The database's name. Pass a pointer to 32-byte character array for this parameter, or `NULL` if you don't care about the name.

← *attributesP*
The database's attribute flags. The section "Database Attributes" lists constants you can use to query the values

returned in this parameter. Pass `NULL` for this parameter if you don't want to retrieve it.

← *`versionP`*
The application-specific version number. The default version number is 0. Pass `NULL` for this parameter if you don't want to retrieve it.

← *`crDateP`*
The date the database was created, expressed as the number of seconds since the first instant of Jan. 1, 1904. Pass `NULL` for this parameter if you don't want to retrieve it.

← *`modDateP`*
The date the database was last modified, expressed as the number of seconds since the first instant of Jan. 1, 1904. Pass `NULL` for this parameter if you don't want to retrieve it.

← *`bckUpDateP`*
The date the database was backed up, expressed as the number of seconds since the first instant of Jan. 1, 1904. Pass `NULL` for this parameter if you don't want to retrieve it.

← *`modNumP`*
The modification number, which is incremented every time a record in the database is added, modified, or deleted. Pass `NULL` for this parameter if you don't want to retrieve it.

← *`appInfoIDP`*
The local ID of the application info block, or `NULL`. The application info block is an optional field that the database may use to store application-specific information about the database. Pass `NULL` for this parameter if you don't want to retrieve it.

← *`sortInfoIDP`*
The local ID of the database's sort table. This is an optional field in the database header. Pass `NULL` for this parameter if you don't want to retrieve it.

← *`typeP`*
The database's type, specified when it is created. Pass `NULL` for this parameter if you don't want to retrieve it.

← *`creatorP`*
The database's creator, specified when it is created. Pass `NULL` for this parameter if you don't want to retrieve it.

Returns Returns `errNone` if no error, or `dmErrInvalidParam` if an error occurs.

Comments The modification date is updated only if a change has been made to the database opened with write access. (The update still occurs upon closing the database.) Changes that trigger an update include adding, deleting, archiving, rearranging, or resizing records, setting a record's dirty bit in `DmReleaseRecord()`, rearranging or deleting categories, or updating the database header fields using `DmSetDatabaseInfo()`.

Compatibility This function is provided for compatibility purposes only; Palm OS Cobalt applications will likely want to use `DmDatabaseInfo()` instead.

See Also `DmDatabaseInfo()`, `DmSetDatabaseInfo()`, `DmDatabaseSize()`, `DmOpenDatabaseInfoV50()`, `DmFindDatabase()`, `DmGetNextDatabaseByTypeCreator()`, `TimSecondsToDateTime()`

DmDatabaseProtectV50 Function

Purpose Increment or decrement a non-schema database's protection count.

Declared In `DataMgr.h`

Prototype

```
status_t DmDatabaseProtectV50 (uint16_t cardNo,
   LocalID dbID, Boolean protect)
```

Parameters

→ *`cardNo`*
Card number of database to protect/unprotect.

→ *`dbID`*
Local ID of database to protect/unprotect.

→ *`protect`*
If `true`, the database's protection count is incremented. If `false`, it is decremented.

Returns Returns `errNone` if no error, or one of the following if an error occurs:

`memErrCardNotPresent`
The specified card can't be found.

`dmErrROMBased`
: You've attempted to delete or modify a ROM-based database.

`dmErrCantFind`
: The specified database can't be found.

`memErrNotEnoughSpace`
: A memory error occurred.

`dmErrDatabaseNotProtected`

Comments This function can be used to prevent a database from being deleted (by passing `true` for the *`protect`* parameter). It increments the protect count if *`protect`* is `true` and decrements it if *`protect`* is `false`. All `true` calls should be balanced by `false` calls before the application terminates.

Use this function if you want to keep a particular record or resource in a database locked down but don't want to keep the database open. This information is kept in the dynamic heap, so all databases are "unprotected" at system reset.

If the database is a resource database that has an overlay associated with it for the current locale, the overlay is also protected or unprotected by this call.

Compatibility This function is provided for compatibility purposes only. Palm OS Cobalt functions should use `DmSetDatabaseProtection()` instead.

DmDatabaseSize Function

Purpose Retrieve size information for a database.

Declared In `DataMgr.h`

Prototype

```
status_t DmDatabaseSize (DatabaseID dbID,
   uint32_t *numRecordsP, uint32_t *totalBytesP,
   uint32_t *dataBytesP)
```

Parameters

→ *`dbID`*
: Database ID of the database.

← *numRecordsP*
: The total number of records in the database. Pass `NULL` for this parameter if you don't want to retrieve it.

← *totalBytesP*
: The total number of bytes used by the database including the overhead. Pass `NULL` for this parameter if you don't want to retrieve it.

← *dataBytesP*
: The total number of bytes used to store just each record's data, not including overhead. Pass `NULL` for this parameter if you don't want to retrieve it.

Returns Returns `errNone` if no error, or `dmErrMemError` if an error occurs.

Comments This function operates on extended, classic, or schema databases.

See Also DmDatabaseInfo(), DmOpenDatabaseInfoV50(), DmFindDatabase(), DmGetNextDatabaseByTypeCreator()

DmDatabaseSizeV50 Function

Purpose Retrieve size information for a database.

Declared In `DataMgr.h`

Prototype

```
status_t DmDatabaseSizeV50 (uint16_t cardNo,
   LocalID dbID, uint32_t *numRecordsP,
   uint32_t *totalBytesP, uint32_t *dataBytesP)
```

Parameters

→ *cardNo*
: Card number the database resides on.

→ *dbID*
: Database ID of the database.

← *numRecordsP*
: The total number of records in the database. Pass `NULL` for this parameter if you don't want to retrieve it.

← *totalBytesP*
: The total number of bytes used by the database including the overhead. Pass `NULL` for this parameter if you don't want to retrieve it.

← *`dataBytesP`*
The total number of bytes used to store just each record's data, not including overhead. Pass `NULL` for this parameter if you don't want to retrieve it.

Returns Returns `errNone` if no error, or `dmErrMemError` if an error occurs.

Compatibility This function is provided for compatibility purposes only. Palm OS Cobalt applications should use `DmDatabaseSize()` instead.

See Also DmDatabaseInfo(), DmOpenDatabaseInfoV50(), DmFindDatabase(), DmGetNextDatabaseByTypeCreator()

DmDeleteCategory Function

Purpose Delete all records in a category. The category name is not changed.

Declared In `DataMgr.h`

Prototype

```
status_t DmDeleteCategory (DmOpenRef dbRef,
   uint16_t categoryNum)
```

Parameters

→ *`dbRef`*
`DmOpenRef` to an open database.

→ *`categoryNum`*
Category of records to delete. Category masks such as `dmAllCategories` are invalid.

Returns Returns `errNone` if no error, or one of the following if an error occurs:

`dmErrReadOnly`
You've attempted to write to or modify a database that is open in read-only mode.

`memErrInvalidParam`
A memory error occurred.

Some releases may display a fatal error message instead of returning the error code.

Comments This function deletes all records in a category, but does not delete the category itself (note that it deletes the record data and header info, and doesn't just set the deleted bit). For each record in the category, `DmDeleteCategory()` marks the `delete` bit in the database header for the record and disposes of the record's data chunk. The record entry in the database header remains, but its `localChunkID` is set to `NULL`.

If the category contains no records, this function does nothing and returns `errNone` to indicate success. The *`categoryNum`* parameter is assumed to represent a single category. If you pass a category mask to specify more than one category, this function interprets that value as a single category, finds no records to delete in that category, and returns `errNone`.

Example You can use the `DmGetRecordCategory()` call to obtain a category index from a given record, as shown in the following code excerpt:

```
DmOpenRef myDB;   //assume that this is set
uint16_t myRecIndex;   //assume that this is set
uint8_t category;
status_t err;

err = DmGetRecordCategory(myDB, myRecIndex, &category);
err = DmDeleteCategory(myDB, category);
```

DmDeleteDatabase Function

Purpose Delete a database and all of its records.

Declared In `DataMgr.h`

Prototype `status_t DmDeleteDatabase (DatabaseID` *`dbID`*`)`

Parameters → *`dbID`*
Database ID of the database being deleted.

Returns Returns `errNone` if no error, or one of the following if an error occurred:

`dmErrCantFind`
The specified database can't be found.

`dmErrCantOpen`
The database cannot be opened.

`memErrChunkLocked`
The associated memory chunk is locked.

`dmErrDatabaseOpen`
: The function cannot be performed on an open database, and the database is open.

`dmErrROMBased`
: You've attempted to delete or modify a ROM-based database.

`memErrInvalidParam`
: A memory error occurred.

`memErrNotEnoughSpace`
: A memory error occurred.

Comments Call this function to delete a database. This function deletes the database, the application info block, the sort info block, and any other overhead information that is associated with this database. After deleting the database, this function enqueues a deferred `sysNotifyDBDeletedEvent` notification, which will be broadcast at the top of the event loop.

If the database has an overlay associated with it, this function does *not* delete the overlay. You can delete the overlay with a separate call to `DmDeleteDatabase()`.

This function accepts a database ID as a parameter. To determine the database ID, call `DmFindDatabase()`.

IMPORTANT: When called from the main application thread, this function may block. While blocked, the application will not receive events and won't redraw its windows. As well, deferred sublaunches and notifications won't execute while the main application thread is blocked.

See Also DmDeleteRecord(), DmRemoveRecord(), DmRemoveResource(), DmCreateDatabase(), DmGetNextDatabaseByTypeCreator(), DmFindDatabase()

DmDeleteDatabaseV50 Function

Purpose Delete a database and all its records.

Declared In `DataMgr.h`

Prototype
```
status_t DmDeleteDatabaseV50 (uint16_t cardNo,
   LocalID dbID)
```

Parameters

→ *`cardNo`*
Card number the database resides on.

→ *`dbID`*
Database ID.

Returns Returns `errNone` if no error, or one of the following if an error occurs:

`dmErrCantFind`
The specified database can't be found.

`dmErrCantOpen`
The database cannot be opened.

`memErrChunkLocked`
The associated memory chunk is locked.

`dmErrDatabaseOpen`
The function cannot be performed on an open database, and the database is open.

`dmErrDatabaseProtected`
The database is marked as protected.

`dmErrROMBased`
You've attempted to delete or modify a ROM-based database.

`memErrInvalidParam`
A memory error occurred.

`memErrNotEnoughSpace`
A memory error occurred.

Comments Call this function to delete a database. This function deletes the database, the application info block, the sort info block, and any other overhead information that is associated with this database. After deleting the database, this function enqueues a deferred `sysNotifyDBDeletedEvent` notification, which will be broadcast at the top of the event loop.

If the database has an overlay associated with it, this function does *not* delete the overlay. You can delete the overlay with a separate call to `DmDeleteDatabase()`.

This function accepts a database ID as a parameter. To determine the database ID, call either `DmFindDatabase()` or `DmGetDatabaseV50()` with a database index.

IMPORTANT: When called from the main application thread, this function may block. While blocked, the application will not receive events and won't redraw its windows. As well, deferred sublaunches and notifications won't execute while the main application thread is blocked.

Compatibility This function is provided for compatibility purposes. Palm OS Cobalt applications will likely want to use `DmDeleteDatabase()` instead.

See Also DmDeleteRecord(), DmRemoveRecord(), DmRemoveResource(), DmCreateDatabase(), DmGetNextDatabaseByTypeCreator(), DmFindDatabase()

DmDeleteRecord Function

Purpose Delete a record's chunk from a database but leave the record entry in the header and set the `delete` bit for the next HotSync operation.

Declared In `DataMgr.h`

Prototype

```
status_t DmDeleteRecord (DmOpenRef dbRef,
   uint16_t index)
```

Parameters

→ *`dbRef`*
`DmOpenRef` to an open database.

→ *`index`*
Which record to delete.

Returns Returns `errNone` if no error, or one of the following if an error occurs:

`dmErrReadOnly`
You've attempted to write to or modify a database that is open in read-only mode.

`dmErrIndexOutOfRange`
: The specified index is out of range.

`dmErrRecordArchived`
: The function requires that the record not be archived, but it is.

`dmErrRecordDeleted`
: The record has been deleted.

`memErrInvalidParam`
: A memory error occurred.

Some releases may display a fatal error message instead of returning the error code.

Comments Marks the `delete` bit in the database header for the record and disposes of the record's data chunk. Does not remove the record entry from the database header, but simply sets the `localChunkID` of the record entry to `NULL`.

See Also DmDetachRecord(), DmRemoveRecord(), DmArchiveRecord(), DmNewRecord()

DmDetachRecord Function

Purpose Detach and orphan a record from a database but don't delete the record's chunk.

Declared In `DataMgr.h`

Prototype
```
status_t DmDetachRecord (DmOpenRef dbRef,
   uint16_t index, MemHandle *hDetached)
```

Parameters

→ *`dbRef`*
: `DmOpenRef` to an open database.

→ *`index`*
: Index of the record to detach.

↔ *`hDetached`*
: Pointer to return handle of the detached record.

Returns Returns `errNone` if no error, or one of the following if an error occurs:

`dmErrReadOnly`
: You've attempted to write to or modify a database that is open in read-only mode.

`dmErrIndexOutOfRange`
: The specified index is out of range.

`dmErrNotRecordDB`
: You've attempted to perform a record function on a resource database.

`memErrChunkLocked`
: The associated memory chunk is locked.

`memErrInvalidParam`
: A memory error occurred.

Some releases may display a fatal error message instead of returning the error code.

Comments This function detaches a record from a database by removing its entry from the database header and returns the handle of the record's data chunk in `*`*`hDetached`*. Unlike `DmDeleteRecord()`, this function removes its entry in the database header but it does not delete the actual record.

See Also DmAttachRecord(), DmRemoveRecord(), DmArchiveRecord(), DmDeleteRecord()

DmDetachResource Function

Purpose Detach a resource from a database and return the handle of the resource's data.

Declared In `DataMgr.h`

Prototype
```
status_t DmDetachResource (DmOpenRef dbRef,
   uint16_t index, MemHandle *hDetached)
```

Parameters

→ *`dbRef`*
: `DmOpenRef` to an open database.

→ *`index`*
: Index of resource to detach.

↔ *`hDetached`*
: Pointer to return handle of the detached record.

Returns Returns `errNone` if no error, or one of the following if an error occurs:

`dmErrReadOnly`
: You've attempted to write to or modify a database that is open in read-only mode.

`dmErrIndexOutOfRange`
: The specified index is out of range.

`dmErrCorruptDatabase`
: The database is corrupted.

`memErrChunkLocked`
: The associated memory chunk is locked.

`memErrInvalidParam`
: A memory error occurred.

Some releases may display a fatal error message instead of returning the error code. All releases may display a fatal error message if the database is not a resource database.

Comments This function detaches a resource from a database by removing its entry from the database header and returns the handle of the resource's data chunk in *`*hDetached`*.

See Also DmAttachResource(), DmRemoveResource()

DmFindDatabase Function

Purpose Return the database ID of a database given its name and creator ID.

Declared In `DataMgr.h`

Prototype
```
DatabaseID DmFindDatabase (const char *nameP,
   uint32_t creator, DmFindType find,
   DmDatabaseInfoPtr databaseInfoP)
```

Parameters

→ *`nameP`*
: Name of the database to look for.

→ *`creator`*
: Creator ID of the database to look for.

→ *find*
Flags indicating the type of database to be searched for: schema, extended, classic, or a combination of the three. See `DmFindType` for more information.

← *databaseInfoP*
Pointer to a `DmDatabaseInfoType` structure which is filled out appropriately for the found database, or `NULL` if this information isn't needed.

Returns Returns the database ID. If the database can't be found, this function returns 0, and `DmGetLastErr()` returns an error code indicating the reason for failure.

Comments This function first searches in RAM; if a database matching the specified criteria is not found, it then searches the device's ROM.

See Also DmDatabaseInfo(), DmFindDatabaseByTypeCreator(), DmFindDatabaseV50(), DmGetNextDatabaseByTypeCreator()

DmFindDatabaseByTypeCreator Function

Purpose Return the database ID of a database given its type and creator ID.

Declared In `DataMgr.h`

Prototype

```
DatabaseID DmFindDatabaseByTypeCreator
   (uint32_t type, uint32_t creator,
   DmFindType find,
   DmDatabaseInfoPtr databaseInfoP)
```

Parameters

→ *type*
Database type of the database to look for.

→ *creator*
Creator ID of the database to look for.

→ *find*
Flags indicating the type of database to be searched for: schema, extended, classic, or a combination of the three. See `DmFindType` for more information.

← *databaseInfoP*
Pointer to a `DmDatabaseInfoType` structure which is filled out appropriately for the found database, or `NULL` if this information isn't needed.

Returns Returns the database ID. If the database can't be found, this function returns 0, and `DmGetLastErr()` returns an error code indicating the reason for failure.

Comments This function first searches in RAM; if a database matching the specified criteria is not found, it then searches the device's ROM.

This function can be used to find extended, classic, or even schema databases.

See Also DmDatabaseInfo(), DmFindDatabase(), DmGetNextDatabaseByTypeCreator()

DmFindDatabaseV50 Function

Purpose Return the database ID of a classic database given its card number and name.

Declared In `DataMgr.h`

Prototype

```
LocalID DmFindDatabaseV50 (uint16_t cardNo,
   const char *nameP)
```

Parameters

→ *cardNo*
Number of card to search.

→ *nameP*
Name of the database to look for.

Returns Returns the database ID. If the database can't be found, this function returns 0, and `DmGetLastErr()` returns an error code indicating the reason for failure.

Comments This function searches only within the classic namespace. This eliminates the possibility of finding multiple databases with the same name.

Palm OS Cobalt applications should usually use `DmFindDatabase()` instead of this function. In order to ensure compatibility, this function only searches for classic database. Note that this function isn't as flexible as `DmFindDatabase()` since it finds databases without regard to their creator ID. This is consistent with earlier versions of Palm OS, in which databases had to be uniquely identified by name.

Compatibility This function is provided for compatibility purposes only. Palm OS Cobalt applications should use `DmFindDatabase()` instead.

See Also DmFindDatabase(), DmGetNextDatabaseByTypeCreator(), DmDatabaseInfo(), DmOpenDatabase()

DmFindRecordByID Function

Purpose Return the index of the record with the given unique ID.

Declared In `DataMgr.h`

Prototype

```
status_t DmFindRecordByID (DmOpenRef dbRef,
   uint32_t uniqueID, uint16_t *pIndex)
```

Parameters

→ *dbRef*
`DmOpenRef` to an open database.

→ *uniqueID*
Unique ID to search for.

← *pIndex*
Return index.

Returns Returns 0 if found, otherwise `dmErrUniqueIDNotFound`. May display a fatal error message if the unique ID is invalid.

See Also DmQueryRecord(), DmGetRecord(), DmRecordInfoV50()

DmFindRecordByOffsetInCategory Function

Purpose Return the index of the record nearest the offset from the passed record index whose category matches the passed category. (The *offset* parameter indicates the number of records to move forward or backward.)

Declared In `DataMgr.h`

Prototype

```
status_t DmFindRecordByOffsetInCategory
   (DmOpenRef dbRef, uint16_t *pIndex,
   uint16_t offset, int16_t direction,
   uint16_t category)
```

Parameters

→ *dbRef*
`DmOpenRef` to an open database.

↔ *`pIndex`*
The index to start the search at. Upon return, contains the index of the record at *`offset`* from the index that you passed in.

→ *`offset`*
Offset of the passed record index. This must be a positive number; use `dmSeekBackward` for the *`direction`* parameter to search backwards.

→ *`direction`*
Must be either `dmSeekForward` or `dmSeekBackward`.

→ *`category`*
Category index.

Returns Returns `errNone` if no error; returns `dmErrIndexOutOfRange` or `dmErrSeekFailed` if an error occurred.

Comments `DmFindRecordByOffsetInCategory()` searches for a record in the specified category. The search begins with the record at *`pIndex`*. When it finds a record in the specified category, it decrements the *`offset`* parameter and continues searching until a match is found and *`offset`* is 0.

Because of this, if you use `DmFindRecordByOffsetInCategory()` to find the nearest matching record in a particular category, you must pass different *`offset`* parameters if the starting record is in the category than if it isn't. If the record at *`pIndex`* is in the category, then you must pass an *`offset`* of 1 to find the next record in the category because the comparison is performed before the *`pIndex`* value changes. If the record at *`pIndex`* isn't in the category, you must pass an *`offset`* of 0 to find the next record in the category. In this case, an *`offset`* of 1 skips the first matching record.

Records that have the `deleted` bit set are ignored, and if the user has specified that private records should be hidden or masked, private records are ignored as well.

See Also DmNumRecordsInCategory(), DmQueryNextInCategory(), DmMoveCategory()

DmFindResource Function

Purpose Search the given database for a resource by type and ID, or by pointer if it is non-`NULL`.

Declared In `DataMgr.h`

Prototype

```
uint16_t DmFindResource (DmOpenRef dbRef,
    DmResourceType resType, DmResourceID resID,
    MemHandle hResource)
```

Parameters

→ *`dbRef`*
  `DmOpenRef` to an open database.

→ *`resType`*
  Type of resource to search for.

→ *`resID`*
  ID of resource to search for.

→ *`hResource`*
  Pointer to locked resource, or `NULL`.

Returns Returns index of resource in resource database, or `dmInvalidRecIndex` if not found.

May display a fatal error message if the database is not a resource database.

Comments Use this function to find a resource in a particular resource database by type and ID or by pointer. It is particularly useful when you want to search only one database for a resource and that database is not the topmost one.

IMPORTANT: This function searches for the resource only in the database you specify. If you pass a pointer to a base resource database, its overlay is *not* searched. To search both a base database and its overlay for a localized resource, use `DmGet1ResourceV50()` instead of this function.

If *`hResource`* is `NULL`, the resource is searched for by type and ID.

If *`hResource`* is not `NULL`, *`resType`* and *`resID`* are ignored and the index of the given locked resource is returned.

Once the index of a resource is determined, it can be locked down and accessed by calling `DmGetResourceByIndex()`.

See Also DmGetResource(), DmSearchResourceOpenDatabases(), DmResourceInfo(), DmGetResourceByIndex(), DmFindResourceType()

DmFindResourceType Function

Purpose Search the given database for a resource by type and type index.

Declared In `DataMgr.h`

Prototype
```
uint16_t DmFindResourceType (DmOpenRef dbRef,
   DmResourceType resType, uint16_t typeIndex)
```

Parameters

→ *`dbRef`*
`DmOpenRef` to an open database.

→ *`resType`*
Type of resource to search for.

→ *`typeIndex`*
Index of given resource type.

Returns Index of resource in resource database, or `0xFFFF` if not found.

May display a fatal error message if the database is not a resource database.

Comments Use this function to retrieve all the resources of a given type in a resource database. By starting at *`typeIndex`* 0 and incrementing until an error is returned, the total number of resources of a given type and the index of each of these resources can be determined. Once the index of a resource is determined, it can be locked down and accessed by calling `DmGetResourceByIndex()`.

IMPORTANT: This function searches for resources only in the database you specify. If you pass a pointer to a base resource database, its overlay is *not* searched. To search both a base database and its overlay for a localized resource, use `DmGet1ResourceV50()` instead of this function.

See Also DmGetResource(), DmSearchResourceOpenDatabases(), DmResourceInfo(), DmGetResourceByIndex(), DmFindResource()

DmGet1ResourceV50 Function

Purpose Search the most recently opened resource database and return a handle to a resource given the resource type and ID.

Declared In `DataMgr.h`

Prototype

```
MemHandle DmGet1ResourceV50
   (DmResourceType resType, DmResourceID resID)
```

Parameters

→ *resType*
The resource type.

→ *resID*
The resource ID.

Returns Handle to resource data. If unsuccessful, this function returns `NULL` and `DmGetLastErr()` returns an error code indicating the reason for failure.

Comments Searches the most recently opened resource database for a resource of the given type and ID. If the database has an overlay associated with it, the overlay is searched first, and then the base database is searched if the overlay does not contain the resource. If found, the resource handle is returned. The application should call `DmReleaseResource()` as soon as it finishes accessing the resource data. The resource handle is not locked by this function.

Compatibility This function is provided for compatibility purposes only. Palm OS Cobalt applications should use `DmGetResource()` or `DmGetResourceByIndex()` instead.

See Also DmGetResource(), DmReleaseResource(), ResLoadConstant()

DmGetAppInfo Function

Purpose Return the handle of the specified database's application info block.

Declared In `DataMgr.h`

Prototype
```
status_t DmGetAppInfo (DmOpenRef dbRef,
    MemHandle *pAppInfoHandle)
```

Parameters

→ *`dbRef`*
`DmOpenRef` to an open database.

← *`pAppInfoHandle`*
Memory handle of the application info block.

Returns Returns `errNone` if the handle was returned successfully, or one of the following if an error occurred:

`dmErrMemError`
A memory error occurred.

`memErrInvalidParam`
A memory error occurred.

Compatibility This function can be used with extended or classic databases. Note that schema databases don't have an explicit application info block.

DmGetAppInfoIDV50 Function

Purpose Return the local ID of the specified database's application info block.

Declared In `DataMgr.h`

Prototype
```
LocalID DmGetAppInfoIDV50 (DmOpenRef dbRef)
```

Parameters

→ *`dbRef`*
`DmOpenRef` to an open database.

Returns Returns local ID of the application info block. The application info block is an optional field that the database may use to store application-specific information about the database; if the database doesn't have an application info block, `DmGetAppInfoIDV50()` returns zero.

Compatibility This function is provided for compatibility purposes only. Palm OS Cobalt applications will likely want to use `DmGetAppInfo()` instead.

See Also DmDatabaseInfo(), DmOpenDatabase()

DmGetDatabaseLockState Function

Purpose Return information about the number of locked and busy records in a RAM-based non-schema database.

Declared In `DataMgr.h`

Prototype
```
status_t DmGetDatabaseLockState (DmOpenRef dbRef,
   uint8_t *pHighest, uint32_t *pCount,
   uint32_t *pBusy)
```

Parameters

→ *`dbRef`*
: `DmOpenRef` to an open database.

← *`pHighest`*
: The highest lock count found for all of the records in the database. If a database has two records, one has a lock count of 2 and one has a lock count of 1, the highest lock count is 2. Pass `NULL` for this parameter if you don't want to retrieve it.

← *`pCount`*
: The number of records that have the lock count that is returned in the *`pHighest`* parameter. Pass `NULL` for this parameter if you don't want to retrieve it.

← *`pBusy`*
: The number of records that have the busy bit set. Pass `NULL` for this parameter if you don't want to retrieve it.

Returns Returns `errNone` if the operation completed successfully, or one of the following otherwise:

`dmErrInvalidParam`
: *`dbRef`* doesn't reference an open database, or *`dbRef`* references a schema database.

`memErrInvalidParam`
: A memory error occurred.

Comments This function is intended to be used for debugging purposes. You can use it to obtain information about how many records are busy and how much locking occurs.

Because databases stored in ROM cannot be locked, if this function is used with a ROM-based database it returns `errNone` but `*`*`pHighest`*, `*`*`pCount`*, and `*`*`pBusy`* (if supplied) are all set to zero.

DmGetDatabaseV50 Function

Purpose Get the database header ID of a database, given its index and card number.

Declared In `DataMgr.h`

Prototype
```
LocalID DmGetDatabaseV50 (uint16_t cardNo,
    uint16_t index)
```

Parameters

→ *`cardNo`*
Card number of database.

→ *`index`*
Index of database.

Returns Returns the database ID, or 0 if an invalid parameter is passed.

Comments Call this function to retrieve the database ID of a database by index. The index should range from 0 to `DmNumDatabases()`-1.

This function is useful for getting a directory of all databases on a card. The databases returned may reside in either the ROM or the RAM. The order in which databases are returned is not fixed; therefore, you should not rely on receiving a list of databases in a particular order.

Compatibility This function is provided for compatibility purposes. Palm OS Cobalt applications that want to iterate through all of a handheld's databases should use `DmGetNextDatabaseByTypeCreator()` instead.

See Also DmOpenDatabase(), DmNumDatabases(), DmDatabaseInfo(), DmDatabaseSize()

DmGetFallbackOverlayLocale Function

Purpose Get the fallback overlay locale: the locale used when the Data Manager attempts to open an overlay locale for which no valid overlay exists.

Declared In `DataMgr.h`

Prototype
```
status_t DmGetFallbackOverlayLocale
   (LmLocaleType *fallbackLocale)
```

Parameters ← *`fallbackLocale`*
Pointer to a structure into which the fallback overlay locale is written.

Returns Returns `errNone` if the fallback locale was obtained successfully, or `dmErrInvalidParam` if the *`fallbackLocale`* parameter is invalid.

Comments The fallback overlay locale is used by the Data Manager when it attempts to automatically open an overlay using the overlay locale, but no valid overlay exists, and the base probably has been stripped.

See Also `DmGetOverlayDatabaseLocale()`, `DmGetOverlayLocale()`, `DmSetFallbackOverlayLocale()`

DmGetLastErr Function

Purpose Return error code from last Data Manager call.

Declared In `DataMgr.h`

Prototype `status_t DmGetLastErr (void)`

Parameters None.

Returns Error code from last unsuccessful Data Manager call.

Comments Use this function to determine why a Data Manager call failed. In particular, calls like `DmGetRecord()` return 0 if unsuccessful, so calling `DmGetLastErr()` is the only way to determine why they failed.

Note that `DmGetLastErr()` does not always reflect the error status of the last Data Manager call. Rather, it reflects the error status of Data Manager calls that don't return an error code. For some of those calls, the saved error code value is not set to 0 when the call is successful.

For example, if a call to `DmOpenDatabaseByTypeCreator()` returns `NULL` for database reference (that is, it fails), `DmGetLastErr()` returns something meaningful; otherwise, it returns the error value of some previous Data Manager call.

Only the Data Manager functions listed in Table 4.1 currently affect the value returned by `DmGetLastErr()`.

Table 4.1 Functions that affect the value returned by DmGetLastErr()

`DbCursorGetRowCount()`	`DbCursorIsBOF()`
`DbCursorIsDeleted()`	`DbCursorIsEOF()`
`DbHasTable()`	`DbOpenDatabase()`
`DbOpenDatabaseByName()`	`DmFindDatabase()`
`DmFindDatabaseByTypeCreator()`	`DmFindDatabaseV50()`
`DmFindRecordByOffsetInCategory()`	`DmFindResource()`
`DmFindResourceType()`	`DmGetAppInfoIDV50()`
`DmGetDatabaseV50()`	`DmGetPositionInCategory()`
`DmGetRecord()`	`DmGetResource()`
`DmGetResourceByIndex()`	`DmGetResourceV50()`
`DmGet1ResourceV50()`	`DmNewHandle()`
`DmNewRecord()`	`DmNewResource()`
`DmNextOpenDatabase()`	`DmNextOpenDatabaseV50()`
`DmNextOpenResDatabase()`	`DmNextOpenResDatabaseV50()`
`DmNumDatabases()`	`DmNumDatabasesV50()`
`DmNumRecords()`	`DmNumRecordsInCategory()`
`DmNumResources()`	`DmOpenDatabase()`
`DmOpenDatabaseByTypeCreator()`	`DmOpenDBNoOverlay()`

Table 4.1 Functions that affect the value returned by DmGetLastErr() ***(continued)***

`DmQueryNextInCategory()`	`DmQueryRecord()`
`DmResizeRecord()`	`DmResizeResource()`
`DmSearchRecordOpenDatabases()`	`DmSearchResourceOpenDatabases()`

DmGetNextDatabaseByTypeCreator Function

Purpose Iterate to the next database that meets the criteria set forth in a previous call to `DmOpenIteratorByTypeCreator()`.

Declared In `DataMgr.h`

Prototype

```
status_t DmGetNextDatabaseByTypeCreator
   (DmSearchStatePtr stateInfoP,
   DatabaseID *dbIDP,
   DmDatabaseInfoPtr databaseInfoP)
```

Parameters

→ *`stateInfoP`*
Pointer to the `DmSearchStateType` structure originally supplied to `DmOpenIteratorByTypeCreator()`.

← *`dbIDP`*
Pointer to a location into which the ID of the found database is written (a value of zero is written if a database meeting the specified criteria isn't found). Pass `NULL` if the ID of the database isn't needed.

← *`databaseInfoP`*
Pointer to a `DmDatabaseInfoType` structure which is filled out appropriately for the found database. Pass `NULL` if this information isn't needed.

Returns Returns `errNone` if a database meeting the specified criteria is found, `dmErrCantFind` if there are no additional databases meeting the specified criteria, or one of the following if an error occurred:

`dmErrInvalidParam`
The *`find`* parameter passed to `DmOpenIteratorByTypeCreator()` did not contain at least one of the defined database type flags.

Comments Both *dbIDP* and *databaseInfoP* are optional; pass NULL for both if you only need to know if there exists a database that meets your particular criteria. Otherwise, pass pointers as appropriate for one or both.

This function searches all heaps for a match.

To start the search, allocate a DmSearchStateType structure and pass it as the *stateInfoP* parameter in a call to DmOpenIteratorByTypeCreator(). Then, call DmGetNextDatabaseByTypeCreator(). Note that you need to call this function repeatedly to discover all databases having a specified type/creator pair. Finally, be sure to call DmCloseIteratorByTypeCreator() to finalize the iteration.

You can pass dmSearchWildcardID for the *type* or *creator* parameter to conduct searches of wider scope. If the *type* parameter is dmSearchWildcardID, this function can be called successively to return all databases of the given creator. If the *creator* parameter is dmSearchWildcardID, this function can be called successively to return all databases of the given type. You can also pass dmSearchWildcardID as the value for both of these parameters to return all available databases without regard to type or creator.

Because databases are scattered freely throughout memory space, they are not returned in any particular order—any database matching the specified type/creator criteria can be returned. Thus, if the value of the *onlyLatestVers* parameter is false, this function may return a database which is not the most recent version matching the specified type/creator pair. To obtain only the latest version of a database matching the search criteria, set the value of the *onlyLatestVers* parameter to true.

When determining which is the latest version of the database, RAM databases are considered newer than ROM databases that have the same version number. Because of this, you can replace any ROM-based application with your own version of it.

If *onlyLatestVers* is true, you only receive one matching database for each type/creator pair. Note that the behavior is different only when you have specified a value for both *type* and *creator* and *onlyLatestVers* is true.

Example The following code excerpt illustrates how to iterate through the latest versions of all schema databases on the device that have a given type and creator.

```
status_t err;
DmSearchStateType state;
DatabaseID dbID = NULL;
uint32_t creator;
char name[dmDBNameLength];
DmDatabaseInfoType databaseInfo;

// Initialize the DmDatabaseInfoType structure
memset(&databaseInfo, 0x0, sizeof(DmDatabaseInfoType));
databaseInfo.name = name;
databaseInfo.creator = &creator;

err = DmOpenIteratorByTypeCreator(&state, myType, myCreator,
   true, dmHdrAttrSchema);
while (err == errNone) {
   err = DmGetNextDatabaseByTypeCreator(&state, &dbID,
      &databaseInfo);
   if (err == errNone) {
      // a database was found; the ID is in dbID, and info
      // about the database is in databaseInfo. Do something
      // with this information here.
   }
}
DmCloseIteratorByTypeCreator(&state);
```

See Also DmFindDatabase(), DmFindDatabaseByTypeCreator(), DmOpenIteratorByTypeCreator(), DmCloseIteratorByTypeCreator()

DmGetNextDatabaseByTypeCreatorV50 Function

Purpose Return the header ID and card number for a classic database or an extended resource database given the type, the creator, or both. This function searches all heaps for a match.

Declared In `DataMgr.h`

Prototype
```
status_t DmGetNextDatabaseByTypeCreatorV50
    (Boolean newSearch,
    DmSearchStatePtr stateInfoP, uint32_t type,
    uint32_t creator, Boolean onlyLatestVers,
    uint16_t *cardNoP, LocalID *dbIDP)
```

Parameters

→ *`newSearch`*
`true` if starting a new search.

↔ *`stateInfoP`*
If *`newSearch`* is `false`, this must point to the same data used for the previous invocation.

→ *`type`*
Type of database to search for. Pass `dmSearchWildcardID` to find databases with any type.

→ *`creator`*
Creator of database to search for. Pass `dmSearchWildcardID` to find databases with any creator.

→ *`onlyLatestVers`*
If `true`, only the latest version of a database with a given type and creator is returned.

← *`cardNoP`*
On exit, the card number of the found database. Pass `NULL` if you don't need the card number (note that as in Palm OS Cobalt the card number is always zero).

← *`dbIDP`*
Local ID of the found database. Pass `NULL` if you don't need the database's local ID.

Returns Returns `errNone` if no error, or `dmErrCantFind` if no matches were found.

Comments You may need to call this function successively to discover all databases having a specified type/creator pair.

To start the search, pass `true` for *`newSearch`*. Allocate a `DmSearchStateType` structure and pass it as the *`stateInfoP`* parameter. `DmGetNextDatabaseByTypeCreator()` stores private information in *`stateInfoP`* and uses it if the search is continued.

To continue a search where the previous one left off, pass `false` for *`newSearch`* and pass the same *`stateInfoP`* that you used during the previous call to this function.

You can pass `dmSearchWildcardID` for the *`type`* or *`creator`* parameter to conduct searches of wider scope. If the *`type`* parameter is `dmSearchWildcardID`, this function can be called successively to return all databases of the given creator. If the *`creator`* parameter is `dmSearchWildcardID`, this function can be called successively to return all databases of the given type. You can also pass `dmSearchWildcardID` as the value for both of these parameters to return all available databases without regard to type or creator.

Because databases are scattered freely throughout memory space, they are not returned in any particular order—any database matching the specified type/creator criteria can be returned.Thus, if the value of the *`onlyLatestVers`* parameter is `false`, this function may return a database which is not the most recent version matching the specified type/creator pair. To obtain only the latest version of a database matching the search criteria, set the value of the *`onlyLatestVers`* parameter to `true`.

When determining which is the latest version of the database, RAM databases are considered newer than ROM databases that have the same version number. Because of this, you can replace any ROM-based application with your own version of it. Also, a RAM database on card 1 is considered newer than a RAM database on card 0 if the version numbers are identical.

WARNING! Don’t create or delete a database while using `DmGetNextDatabaseByTypeCreatorV50()` to iterate through the existing databases. This could cause databases to be skipped, or it could result in a given database being returned more than once.

If *`onlyLatestVers`* is `true`, you only receive one matching database for each type/creator pair. Note that the behavior is different only when you have specified a value for both *`type`* and *`creator`* and *`onlyLatestVers`* is `true`.

If you expect multiple databases to match your search criteria, make sure you call `DmGetNextDatabaseByTypeCreator()` in one of the following ways to ensure that your code operates the same on all Palm OS versions:

- Set *`onlyLatestVers`* to `false` if you specify both a *`type`* and *`creator`*.
- Specify `0` for either the *`type`* or *`creator`* parameter (or both).

Compatibility This function is provided for compatibility purposes only. Most Palm OS Cobalt applications will want to use `DmGetNextDatabaseByTypeCreator()` instead; that function (in conjunction with `DmOpenIteratorByTypeCreator()` and `DmCloseIteratorByTypeCreator()`) can be used to locate classic, extended, or schema databases.

See Also DmFindDatabase(), DmDatabaseInfo(), DmOpenDatabaseByTypeCreator(), DmDatabaseSize()

DmGetOpenInfo Function

Purpose Retrieve information about an open database.

Declared In `DataMgr.h`

Prototype

```
status_t DmGetOpenInfo (DmOpenRef dbRef,
   DatabaseID *pDbID, uint16_t *pOpenCount,
   DmOpenModeType *pOpenMode, Boolean *pResDB)
```

Parameters

→ *`dbRef`*
`DmOpenRef` to an open database.

← *`pDbID`*
ID of the database. Pass `NULL` for this parameter if you don’t want to retrieve this information.

← *pOpenCount*
: Number of applications that have this database open. Pass `NULL` for this parameter if you don't want to retrieve this information.

← *pOpenMode*
: Mode used to open the database (see `DmOpenModeType`). Pass `NULL` for this parameter if you don't want to retrieve this information.

← *pResDB*
: If `true` upon return, the database is a resource database. Otherwise, the database is a record database. Pass `NULL` for this parameter if you don't want to retrieve this information.

Returns Returns `errNone` if no error.

See Also `DmDatabaseInfo()`

DmGetOverlayDatabaseLocale Function

Purpose Return an overlay database's locale given its name.

Declared In `DataMgr.h`

Prototype

```
status_t DmGetOverlayDatabaseLocale
    (const char *overlayDBName,
    LmLocaleType *overlayLocale)
```

Parameters

→ *overlayDBName*
: The name of the overlay database.

← *overlayLocale*
: Points to an `LmLocaleType` structure into which the overlay's locale is written. Your application must allocate and pass a pointer to this structure.

Returns Returns `errNone` upon success, or one of the following if an error occurred:

`dmErrInvalidParam`
: The function received an invalid parameter.

`dmErrBadOverlayDBName`
: The *overlayDBName* parameter doesn't point to the name of an overlay database.

DmGetOverlayDatabaseName Function

Purpose Return the overlay database's name given the base database name and the locale.

Declared In `DataMgr.h`

Prototype
```
status_t DmGetOverlayDatabaseName
    (const char *baseDBName,
    const LmLocaleType *targetLocale,
    char *overlayDBName)
```

Parameters

→ *`baseDBName`*
The name of the base database with which the overlay is associated.

→ *`targetLocale`*
The locale to which this overlay applies. See `LmLocaleType`. Pass `NULL` to use the current locale.

← *`overlayDBName`*
Pointer to a buffer into which the overlay database name is written. This buffer must be at least `dmDBNameLength` bytes.

Returns Returns `errNone` upon success, or `dmErrInvalidParam` if one of the parameters is invalid.

DmGetOverlayLocale Function

Purpose Get the Data Manager's overlay locale: the locale used by the Data Manager when it attempts to automatically open overlays.

Declared In `DataMgr.h`

Prototype
```
status_t DmGetOverlayLocale
    (LmLocaleType *overlayLocale)
```

Parameters

← *`overlayLocale`*
Pointer to an `LmLocaleType` structure into which the overlay's locale is written. Your application must allocate and pass a pointer to this structure.

Returns Returns `errNone` upon success, or `dmErrInvalidParam` if one of the parameters is invalid.

See Also `DmGetOverlayDatabaseLocale()`, `DmSetOverlayLocale()`

DmGetPositionInCategory Function

Purpose Return a position of a record within the specified category.

Declared In `DataMgr.h`

Prototype
```
uint16_t DmGetPositionInCategory
    (DmOpenRef dbRef, uint16_t index,
    uint16_t category)
```

Parameters

→ *`dbRef`*
`DmOpenRef` to an open database.

→ *`index`*
Index of the record.

→ *`category`*
Index of category to search.

Returns Returns the position (zero-based). If the specified index is out of range, this function returns 0 and `DmGetLastErr()` returns an error code indicating the reason for failure. Note that this means a 0 return value might indicate either success or failure. If this function returns 0 and `DmGetLastErr()` returns `errNone`, the return value indicates that this is the first record in the category.

Comments Because this function must examine all records up to the current record, it can be slow to return, especially when called on a large database.

Records that have the `deleted` bit set are ignored, and if the user has specified that private records should be hidden or masked, private records are ignored as well.

If the record is ROM-based (pointer accessed) this function makes a fake handle to it and stores this handle in the `DmAccessType` structure.

To learn which category a record is in, use `DmGetRecordCategory()`.

See Also DmQueryNextInCategory(), DmFindRecordByOffsetInCategory(), DmMoveCategory()

DmGetRecord Function

Purpose Return a handle to a record by index and mark the record busy.

Declared In `DataMgr.h`

Prototype
```
MemHandle DmGetRecord (DmOpenRef dbRef,
   uint16_t index)
```

Parameters
→ *dbRef*
  `DmOpenRef` to an open database.

→ *index*
  Which record to retrieve.

Returns Returns a handle to record data. If another call to `DmGetRecord()` for the same record is attempted before the record is released, `NULL` is returned and `DmGetLastErr()` returns an error code indicating the reason for failure.

Comments Returns a handle to given record and sets the `busy` bit for the record.

If the record is ROM-based (pointer accessed), this function makes a fake handle to it and stores this handle in the `DmAccessType` structure.

`DmReleaseRecord()` should be called as soon as the caller finishes viewing or editing the record.

See Also DmSearchRecordOpenDatabases(), DmFindRecordByID(), DmRecordInfoV50(), DmReleaseRecord(), DmQueryRecord()

DmGetRecordAttr Function

Purpose Get the attributes of a database record.

Declared In `DataMgr.h`

Prototype
```
status_t DmGetRecordAttr (DmOpenRef dbRef,
   uint16_t index, uint8_t *pAttr)
```

Parameters
→ *dbRef*
  `DmOpenRef` to an open database.

→ *index*
  Index of the record for which attributes are being retrieved.

← *`pAttr`*
: Pointer to a variable into which the record's attributes are written. See "Non-Schema Database Record Attributes" on page 108 for a description of the attributes.

Returns Returns `errNone` if the attributes were successfully obtained, or one of the following if an error occurred:

`dmErrNotRecordDB`
: You've attempted to perform a record function on a resource database.

`dmErrIndexOutOfRange`
: The specified index is out of range.

See Also `DmRecordInfoV50()`, `DmSetRecordAttr()`

DmGetRecordCategory Function

Purpose Get the category information for a record.

Declared In `DataMgr.h`

Prototype
```
status_t DmGetRecordCategory (DmOpenRef dbRef,
   uint16_t index, uint8_t *pCategory)
```

Parameters

→ *`dbRef`*
: `DmOpenRef` to an open database.

→ *`index`*
: Index of the record for which the category information is being obtained.

← *`pCategory`*
: Pointer to a variable into which the record's category information is written.

Returns Returns `errNone` if the category information was successfully obtained, or one of the following if an error occurred:

`dmErrNotRecordDB`
: You've attempted to perform a record function on a resource database.

`dmErrIndexOutOfRange`
: The specified index is out of range.

See Also `DmRecordInfoV50()`, `DmSetRecordCategory()`

DmGetRecordID Function

Purpose Get the record ID for the record at the given index position.

Declared In `DataMgr.h`

Prototype
```
status_t DmGetRecordID (DmOpenRef dbRef,
   uint16_t index, uint32_t *pUID)
```

Parameters

→ *`dbRef`*
: `DmOpenRef` to an open database.

→ *`index`*
: Index of the record for which to retrieve the ID.

← *`pUID`*
: Pointer to a variable into which the record ID is written.

Returns Returns `errNone` if the category information was successfully obtained, or one of the following if an error occurred:

`dmErrNotRecordDB`
: You've attempted to perform a record function on a resource database.

`dmErrIndexOutOfRange`
: The specified index is out of range.

`dmErrInvalidParam`
: The function received an invalid parameter.

See Also `DmRecordInfoV50()`, `DmSetRecordID()`

DmGetRecordSortPosition Function

Purpose Returns where in a sorted list of records a given record would be located. Useful to find where to insert a record with `DmAttachRecord()`. Uses a binary search.

Declared In `DataMgr.h`

Prototype

```
uint16_t DmGetRecordSortPosition
   (DmOpenRef dbRef, void *pNewRecord,
   DmSortRecordInfoType *pNewRecordInfo,
   DmCompareFunctionType *pFuncCompar,
   int16_t other)
```

Parameters

→ *`dbRef`*
  `DmOpenRef` to an open database.

→ *`pNewRecord`*
  Pointer to the new record.

→ *`pNewRecordInfo`*
  Sort information about the new record. See `DmSortRecordInfoType`.

→ *`pFuncCompar`*
  Pointer to comparison function. See `DmCompareFunctionType()`.

→ *`other`*
  Any value the application wants to pass to the comparison function. This parameter is often used to indicate a sort direction (ascending or descending).

Returns The position where the record should be inserted.

The position should be viewed as between the record returned and the record before it. Note that the return value may be one greater than the number of records.

Comments If *`pNewRecord`* has the same key as another record in the database, `DmGetRecordSortPosition()` assumes that *`pNewRecord`* should be inserted after that record. If there are several records with the same key, *`pNewRecord`* is inserted after all of them. For this reason, if you use `DmGetRecordSortPosition()` to search for the location of a record that you know is already in the database, you must subtract 1 from the result. (Be sure to check that the value is not 0.)

If there are deleted records in the database, `DmGetRecordSortPosition()` only works if those records are at the end of the database. `DmGetRecordSortPosition()` always assumes that a deleted record is greater than or equal to any other record.

DmGetResource Function

Purpose Search a specified open database and return a handle to a resource, given the resource type and ID.

Declared In `DataMgr.h`

Prototype

```
MemHandle DmGetResource (DmOpenRef dbRef,
   DmResourceType resType, DmResourceID resID)
```

Parameters

→ *`dbRef`*
Reference to an open database to be searched.

→ *`resType`*
The resource type.

→ *`resID`*
The resource ID.

Returns Handle to resource data. If the specified resource cannot be found, this function returns `NULL` and `DmGetLastErr()` returns an error code indicating the reason for failure.

Comments Searches the specified database for a resource of the given type and ID. If found, the resource handle is returned. The application should call `DmReleaseResource()` as soon as it finishes accessing the resource data. The resource handle is not locked by this function.

This function always returns the resource located in the overlay if the overlay has a resource matching that type and ID. If there is no overlay version of the resource, this function returns the resource from the base database.

See Also DmGet1ResourceV50(), DmReleaseResource(), ResLoadConstant()

DmGetResourceByIndex Function

Purpose Return a handle to a resource, given the index of that resource.

Declared In `DataMgr.h`

Prototype

```
MemHandle DmGetResourceByIndex (DmOpenRef dbRef,
   uint16_t index)
```

Parameters

→ *dbRef*
  `DmOpenRef` to an open database.

→ *index*
  Index of the resource whose handle you want.

Returns Handle to resource data. If the specified index is out of range, this function returns `NULL` and `DmGetLastErr()` returns an error code indicating the reason for failure.

May display a fatal error message if the database is not a resource database.

IMPORTANT: This function accesses the resource only in the database you specify. If you pass a pointer to a base resource database, its overlay is *not* accessed. Therefore, you should use care when using this function to access a potentially localized resource. You can use `DmSearchResourceOpenDatabases()` to obtain a pointer to the overlay database if the resource is localized; however, it's more convenient to use `DmGetResource()` or `DmGet1ResourceV50()`.

See Also DmFindResource(), DmFindResourceType(), DmSearchResourceOpenDatabases()

DmGetResourceV50 Function

Purpose Search all open resource databases and return a handle to a resource, given the resource type and ID.

Declared In `DataMgr.h`

Prototype

```
MemHandle DmGetResourceV50
   (DmResourceType resType, DmResourceID resID)
```

Parameters

→ *`resType`*
The resource type.

→ *`resID`*
The resource ID.

Returns Handle to resource data. If the specified resource cannot be found, this function returns `NULL` and `DmGetLastErr()` returns an error code indicating the reason for failure.

Comments Searches all open resource databases starting with the most recently opened one for a resource of the given type and ID. If found, the resource handle is returned. The application should call `DmReleaseResource()` as soon as it finishes accessing the resource data. The resource handle is not locked by this function.

This function always returns the resource located in the overlay if any open overlay has a resource matching that type and ID. If there is no overlay version of the resource, this function returns the resource from the base database.

Compatibility This function is provided for compatibility purposes. Because most Palm OS Cobalt applications know which resource file should contain the resource being searched for, for efficiency purposes such applications should use `DmGetResource()` or `DmGetResourceByIndex()` instead.

See Also DmGet1ResourceV50(), DmReleaseResource(), ResLoadConstant()

DmGetStorageInfo Function

Purpose Determine how much memory is used, and how much is free, in both secure and non-secure storage.

Declared In `DataMgr.h`

Prototype
```
status_t DmGetStorageInfo
   (DmStorageInfoPtr pStorageInfo)
```

Parameters → *`pStorageInfo`*
Pointer to a `DmStorageInfoType` structure, which upon return contains the memory usage information.

Returns Returns `errNone` if the memory information is obtained successfully, or one of the following otherwise:

`dmErrInvalidParam`
The function received an invalid parameter.

`dmErrMemError`
A memory error occurred.

Comments Your application must allocate the `DmStorageInfoType` structure prior to calling this function.

DmHandleFree Function

Purpose Dispose of a movable chunk on the storage heap.

Declared In `DataMgr.h`

Prototype
```
status_t DmHandleFree (MemHandle handle)
```

Parameters → *`handle`*
Chunk handle.

Returns Returns 0 if no error, or `dmErrInvalidParam` if an error occurred.

Comments Call this function to dispose of a movable chunk.

See Also `MemHandleNew()`

DmHandleLock Function

Purpose Lock a storage heap chunk and obtain a pointer to the chunk's data.

Declared In `DataMgr.h`

Prototype `MemPtr DmHandleLock (MemHandle handle)`

Parameters

→ *`handle`*
Chunk handle.

Returns Returns a pointer to the chunk.

Comments Call this function to lock a chunk and obtain a pointer to it. Call `MemHandleLock()` to lock a chunk allocated on the dynamic heap.

`DmHandleLock()` and `DmHandleUnlock()` should be used in pairs.

See Also `MemHandleNew()`

DmHandleResize Function

Purpose Resize a storage heap chunk.

Declared In `DataMgr.h`

Prototype

```
status_t DmHandleResize (MemHandle handle,
   uint32_t newSize)
```

Parameters

→ *`handle`*
Chunk handle.

→ *`newSize`*
The new desired size.

Returns Returns `errNone` if the chunk was successfully resized, or one of the following if an error occurred:

`dmErrInvalidParam`
Invalid parameter passed.

`memErrNotEnoughSpace`
A memory error occurred.

`memErrChunkLocked`
The associated memory chunk is locked.

Comments Call this function to resize a chunk. This function is always successful when shrinking the size of a chunk, even if the chunk is locked. When growing a chunk, it first attempts to grab free space immediately following the chunk so that the chunk does not have to move. If the chunk has to move to another free area of the heap to grow, it must be movable and have a lock count of 0.

See Also MemHandleNew(), DmHandleSize()

DmHandleSize Function

Purpose Return the requested size of a storage heap chunk.

Declared In `DataMgr.h`

Prototype `uint32_t DmHandleSize (MemHandle` *`handle`*`)`

Parameters → *`handle`*
Chunk handle.

Returns Returns the requested size of the chunk.

Comments Call this function to get the size originally requested for a chunk.

See Also DmHandleResize()

DmHandleUnlock Function

Purpose Unlock a storage heap chunk given a chunk handle.

Declared In `DataMgr.h`

Prototype `status_t DmHandleUnlock (MemHandle` *`handle`*`)`

Parameters → *`handle`*
The chunk handle.

Returns Returns `errNone` if the handle was successfully unlocked, or `dmErrInvalidParam` if the passed handle was invalid.

Comments Call this function to decrement the lock count for a chunk.

`DmHandleLock()` and `DmHandleUnlock()` should be used in pairs.

DmInitiateAutoBackupOfOpenDatabase Function

Purpose Update the automatic backup file for a given open database.

Declared In `DataMgr.h`

Prototype

```
status_t DmInitiateAutoBackupOfOpenDatabase
    (DmOpenRef dbRef)
```

Parameters

→ *`dbRef`*
Database access pointer.

Returns Returns `errNone` if no error, or one of the following if an error occurs:

`dmErrInvalidParam`
`dbRef` doesn't reference a valid open database.

`dmErrReadOnly`
`dbRef` references a non-schema database that is open in read-only mode. Non-schema databases must be open for writing

`dmErrOperationAborted`
The Palm OS device doesn't support the automatic database backup feature.

Comments The database is left open.

Use this function to cause an open database to be backed up.

Many devices running Palm OS Cobalt version 6.1 will back up the contents of the RAM storage heaps to some sort of non-volatile NAND flash. In the event that the RAM storage heaps are corrupted or are lost for some reason, the storage heaps can then be restored to their saved state. Backup is automatically triggered on a limited set of events: database close, database create, a call to `DmSetDatabaseInfo()`, or upon device sleep (open databases only). Developers can explicitly cause a database to be backed up by calling `DmInitiateAutoBackupOfOpenDatabase()`.

For additional information on this feature, see "Automatic Database Backup and Restore" on page 15.

DmInsertionSort Function

Purpose Sort records in a database.

Declared In `DataMgr.h`

Prototype
```
status_t DmInsertionSort (const DmOpenRef dbR,
   DmCompareFunctionType *compar, int16_t other)
```

Parameters

→ *dbR*
Database access pointer.

→ *compar*
Comparison function. See `DmCompareFunctionType()`.

→ *other*
Any value the application wants to pass to the comparison function. This parameter is often used to indicate a sort direction (ascending or descending).

Returns Returns `errNone` if no error, or one of the following if an error occurs:

`dmErrReadOnly`
You've attempted to write to or modify a database that is open in read-only mode.

`dmErrNotRecordDB`
You've attempted to perform a record function on a resource database.

Some releases may display a fatal error message instead of returning the error code.

Comments Deleted records are placed last in any order. All others are sorted according to the passed comparison function. Only records which are out of order move. Moved records are moved to the end of the range of equal records. If a large number of records are being sorted, try to use the quick sort.

The following insertion-sort algorithm is used: Starting with the second record, each record is compared to the preceding record. Each record not greater than the last is inserted into sorted position within those already sorted. A binary insertion is performed. A moved record is inserted after any other equal records.

See Also `DmQuickSort()`

DmMoveCategory Function

Purpose Move all records in a category to another category.

Declared In `DataMgr.h`

Prototype
```
status_t DmMoveCategory (DmOpenRef dbRef,
   uint16_t toCategory, uint16_t fromCategory,
   Boolean fDirty)
```

Parameters

→ *`dbRef`*
  `DmOpenRef` to an open database.

→ *`toCategory`*
  Category to which the records should be added.

→ *`fromCategory`*
  Category from which to remove records.

→ *`fDirty`*
  If `true`, set the dirty bit.

Returns Returns `errNone` if successful, or `dmErrReadOnly` if the database is in read-only mode. Some releases may display a fatal error message instead of returning the error code.

Comments If *`fDirty`* is `true`, the moved records are marked as dirty.

The *`toCategory`* and *`fromCategory`* parameters hold category index values. You can learn which category a record is in with the `DmGetRecordCategory()` call and use that value in this function. For example, the following code, ensures that the records `rec1` and `rec2` are in the same category:

```
DmOpenRef myDB;   //assume that this is set
uint16_t rec1Index, rec2Index;   //assume that these are set
status_t err;
uint8_t category1, category2;

err = DmGetRecordCategory(myDb, rec1Index, &category1);
err = DmGetRecordCategory(myDb, rec2Index, &category2);
if (category1 != category2)
   DmMoveCategory(myDB, category1, category2, true);
```

DmMoveRecord Function

Purpose Move a record from one index to another.

Declared In `DataMgr.h`

Prototype
```
status_t DmMoveRecord (DmOpenRef dbRef,
   uint16_t from, uint16_t to)
```

Parameters

→ *`dbRef`*
: `DmOpenRef` to an open database.

→ *`from`*
: Index of record to move.

→ *`to`*
: Where to move the record.

Returns Returns `errNone` if no error, or one of the following if an error occurs:

`dmErrReadOnly`
: You've attempted to write to or modify a database that is open in read-only mode.

`dmErrIndexOutOfRange`
: The specified index is out of range.

`dmErrNotRecordDB`
: You've attempted to perform a record function on a resource database.

`dmErrMemError`
: A memory error occurred.

`memErrInvalidParam`
: A memory error occurred.

`memErrChunkLocked`
: The associated memory chunk is locked.

Some releases may display a fatal error message instead of returning the error code.

Comments Insert the record at the *`to`* index and move other records down. The *`to`* position should be viewed as an insertion position. This value may be one greater than the index of the last record in the database. In cases where *`to`* is greater than *`from`*, the new index of the record becomes *`to`* – 1 after the move is complete.

DmNewHandle Function

Purpose Attempt to allocate a new chunk in the storage heap.

Declared In `DataMgr.h`

Prototype
```
MemHandle DmNewHandle (DmOpenRef dbRef,
   uint32_t size)
```

Parameters
→ *dbRef*
`DmOpenRef` to an open database.

→ *size*
Size of new handle.

Returns Returns a handle to the new chunk. If an error occurs, returns 0, and `DmGetLastErr()` returns an error code indicating the reason for failure.

Comments Allocates a new handle of the given size. You can attach the handle to the database as a record to obtain and save its record ID in the `appInfoID` or `sortInfoID` fields of the header.

The handle should be attached to a database as soon as possible. If it is not attached to a database and the application crashes, the memory used by the new handle is unavailable until the next soft reset.

DmNewRecord Function

Purpose Return a handle to a new record in the database and mark the record busy.

Declared In `DataMgr.h`

Prototype
```
MemHandle DmNewRecord (DmOpenRef dbRef,
   uint16_t *atP, uint32_t size)
```

Parameters
→ *dbRef*
`DmOpenRef` to an open database.

↔ *atP*
Pointer to index where new record should be placed. Specify the value `dmMaxRecordIndex` to add the record to the end of the database.

→ *`size`*
Size of new record.

Returns Handle to record data. If an error occurs, this function returns 0 and `DmGetLastErr()` returns an error code indicating the reason for failure.

Some releases may display a fatal error message if the database is opened in read-only mode or it is a resource database.

Comments Allocates a new record of the given size, and returns a handle to the record data. The parameter *`atP`* points to an index variable. The new record is inserted at index `*`*`atP`* and all record indices that follow are shifted down. If `*`*`atP`* is greater than the number of records currently in the database, the new record is appended to the end and its index is returned in `*`*`atP`*.

Both the `busy` and `dirty` bits are set for the new record and a unique ID is automatically created.

`DmReleaseRecord()` should be called as soon as the caller finishes viewing or editing the record.

See Also DmAttachRecord(), DmRemoveRecord(), DmDeleteRecord()

DmNewResource Function

Purpose Allocate and add a new resource to a resource database.

Declared In `DataMgr.h`

Prototype
```
MemHandle DmNewResource (DmOpenRef dbRef,
   DmResourceType resType, DmResourceID resID,
   uint32_t size)
```

Parameters

→ *`dbRef`*
`DmOpenRef` to an open database.

→ *`resType`*
Type of the new resource.

→ *`resID`*
ID of the new resource.

→ *`size`*
Desired size of the new resource.

Returns Returns a handle to the new resource. If an error occurs, this function returns `NULL` and `DmGetLastErr()` returns an error code indicating the reason for failure.

May display a fatal error message if the database is not a resource database.

Comments Allocates a memory chunk for a new resource and adds it to the given resource database. The new resource has the given type and ID. If successful, the application should call `DmReleaseResource()` as soon as it finishes initializing the resource.

See Also DmAttachResource(), DmRemoveResource()

DmNextOpenDatabase Function

Purpose Return a `DmOpenRef` to the next open database for the current task.

Declared In `DataMgr.h`

Prototype `DmOpenRef DmNextOpenDatabase (DmOpenRef` *`dbRef`*`)`

Parameters → *`dbRef`*
Current database access pointer or `NULL` to start the search from the top.

Returns `DmOpenRef` to the next open database, or `NULL` if there are no more.

Comments Call this function successively to get the `DmOpenRefs` of all open databases. Pass `NULL` for *`dbRef`* to get the first one. Applications don't usually call this function, but is useful for system information.

Note that unlike `DmNextOpenDatabaseV50()`, this function doesn't find databases that have been added to the resource search chain using functions such as `DmOpenDatabaseV50()`.

IMPORTANT: When called from the main application thread, this function may block. While blocked, the application will not receive events and won't redraw its windows. As well, deferred sublaunches and notifications won't execute while the main application thread is blocked.

See Also DmDatabaseInfo(), DmOpenDatabaseInfoV50()

DmNextOpenDatabaseV50 Function

Purpose Return `DmOpenRef` to the next open database in the current task's search chain.

Declared In `DataMgr.h`

Prototype `DmOpenRef DmNextOpenDatabaseV50 (DmOpenRef` *`dbRef`*`)`

Parameters → *`dbRef`*
Current database access pointer or `NULL` to start the search from the top.

Returns `DmOpenRef` to next open database, or `NULL` if there are no more.

Comments Call this function successively to get the `DmOpenRefs` of all open databases. Pass `NULL` for *`dbRef`* to get the first one. Applications don't usually call this function, but is useful for system information.

This function is provided for backwards compatibility with 68K-based applications. Unlike `DmNextOpenDatabase()`, this function does find databases that have been added to the resource search chain using functions such as `DmOpenDatabaseV50()`.

IMPORTANT: When called from the main application thread, this function may block. While blocked, the application will not receive events and won't redraw its windows. As well, deferred sublaunches and notifications won't execute while the main application thread is blocked.

Compatibility This function—and the concept of a resource search chain—are provided to ease the porting of applications from an earlier version of Palm OS. Palm OS Cobalt applications should use `DmNextOpenDatabase()` instead.

See Also DmDatabaseInfo(), DmOpenDatabaseInfoV50()

DmNextOpenResDatabase Function

Purpose Return an access pointer to next open resource database in the current task.

Declared In `DataMgr.h`

Prototype `DmOpenRef DmNextOpenResDatabase (DmOpenRef dbRef)`

Parameters → *`dbRef`*
Database reference, or `NULL` to start the search from the top.

Returns Pointer to next open resource database.

Comments Returns a pointer to next open resource database. To get a pointer to the first one in the list, pass `NULL` for *`dbRef`*.

If you use this function to access a resource database that might have an overlay associated with it, be careful how you use the result. The `DmOpenRef` returned by this function is a pointer to the overlay database, not the base database. If you subsequently pass this pointer to `DmFindResource()`, you'll receive a handle to the overlay resource. If you're searching for a resource that is found only in the base, you won't find it. Instead, always use `DmGetResource()` or `DmGet1ResourceV50()` to obtain a resource. Both of those functions search both the overlay databases and their associated base databases.

IMPORTANT: When called from the main application thread, this function may block. While blocked, the application will not receive events and won't redraw its windows. As well, deferred sublaunches and notifications won't execute while the main application thread is blocked.

DmNextOpenResDatabaseV50 Function

Purpose Return access pointer to next open resource database in the current task's search chain.

Declared In `DataMgr.h`

Prototype
```
DmOpenRef DmNextOpenResDatabaseV50
   (DmOpenRef dbRef)
```

Parameters
→ *`dbRef`*
Database reference, or 0 to start search from the top.

Returns Pointer to next open resource database.

Comments Returns pointer to next open resource database. To get a pointer to the first one in the search chain, pass `NULL` for *`dbRef`*. This is the database that is searched when `DmGet1ResourceV50()` is called.

If you use this function to access a resource database that might have an overlay associated with it, be careful how you use the result. The `DmOpenRef` returned by this function is a pointer to the overlay database, not the base database. If you subsequently pass this pointer to `DmFindResource()`, you'll receive a handle to the overlaid resource. If you're searching for a resource that is found only in the base, you won't find it. Instead, always use `DmGetResource()` or `DmGet1ResourceV50()` to obtain a resource. Both of those functions search both the overlay databases and their associated base databases.

IMPORTANT: When called from the main application thread, this function may block. While blocked, the application will not receive events and won't redraw its windows. As well, deferred sublaunches and notifications won't execute while the main application thread is blocked.

Compatibility This function—and the concept of a resource search chain—are provided to ease the porting of applications from an earlier version of Palm OS. Palm OS Cobalt applications should use `DmNextOpenResDatabase()` instead.

DmNumDatabases Function

Purpose Determine how many databases reside in memory.

Declared In `DataMgr.h`

Prototype `uint16_t DmNumDatabases (void)`

Parameters None.

Returns The number of databases found.

Comments The returned value doesn't include databases on expansion media (such as an SD card).

See Also `DmGetNextDatabaseByTypeCreator()`

DmNumDatabasesV50 Function

Purpose Determine how many classic databases or extended resource database reside in either RAM or ROM.

Declared In `DataMgr.h`

Prototype `uint16_t DmNumDatabasesV50 (uint16_t` *`cardNo`*`)`

Parameters → *`cardNo`*
Number of the card to check.

Returns The number of databases found.

Comments This function is helpful for getting a directory of all databases on a card. `DmGetDatabaseV50()` accepts an index from 0 to `DmNumDatabases()` -1 and returns a database ID by index.

Compatibility This function only returns the number of classic databases residing in RAM. Palm OS Cobalt applications should use `DmNumDatabases()` instead.

See Also `DmGetDatabaseV50()`

DmNumRecords Function

Purpose Return the number of records in a database.

Declared In `DataMgr.h`

Prototype
```
uint16_t DmNumRecords (DmOpenRef dbRef)
```

Parameters → *`dbRef`*
`DmOpenRef` to an open database.

Returns The number of records in a database.

Comments Records that have that have the `deleted` bit set (that is, records that will be deleted during the next HotSync operation because the user has marked them deleted) are included in the count. If you want to exclude these records from your count, use `DmNumRecordsInCategory()` and pass `dmAllCategories` as the category.

See Also DmNumRecordsInCategory(), DmRecordInfoV50(), DmSetRecordInfoV50()

DmNumRecordsInCategory Function

Purpose Return the number of records of a specified category in a database.

Declared In `DataMgr.h`

Prototype
```
uint16_t DmNumRecordsInCategory (DmOpenRef dbRef,
   uint16_t category)
```

Parameters → *`dbRef`*
`DmOpenRef` to an open database.

→ *`category`*
Category index.

Returns The number of records in the category.

Comments Because this function must examine all records in the database, it can be slow to return, especially when called on a large database.

Records that have the `deleted` bit set are not counted, and if the user has specified to hide or mask private records, private records are not counted either.

You can use the DmGetRecordCategory() call to obtain a category index from a given record. For example:

```
DmOpenRef myDB;   //assume that this is set
uint16_t recIndex;   //assume that this is set
status_t err;
uint8_t category;
uint16_t total;

err = DmGetRecordCategory(myDb, recIndex, &category);
total = DmNumRecordsInCategory(myDB, category);
```

See Also DmNumRecords(), DmQueryNextInCategory(), DmGetPositionInCategory(), DmFindRecordByOffsetInCategory(), DmMoveCategory()

DmNumResources Function

Purpose Return the total number of resources in a given resource database.

Declared In `DataMgr.h`

Prototype `uint16_t DmNumResources (DmOpenRef` *`dbRef`*`)`

Parameters → *dbRef*
`DmOpenRef` to an open database.

Returns The total number of resources in the given database.

May display a fatal error message if the database is not a resource database.

Comments `DmNumResources()` counts only the resources in the database indicated by the `DmOpenRef` parameter. If the database is a resource database that has an overlay associated with it, this function returns only the number of resources in the base database, not in the overlay.

DmOpenDatabase Function

Purpose Open a non-schema database and return a reference to it. If the database is a resource database, also open its overlay for the current locale.

Declared In `DataMgr.h`

Prototype
```
DmOpenRef DmOpenDatabase (DatabaseID dbID,
   DmOpenModeType mode)
```

Parameters → *dbID*
Database ID of the database.

→ *mode*
Which mode to open the database in (see `DmOpenModeType`).

Returns Returns a `DmOpenRef` to the open database. On error, unlike `DmOpenDatabaseV50()`, no fatal error is displayed; this function simply returns 0 and `DmGetLastErr()` returns an error code indicating the reason for failure.

Comments Call this function to open a database for reading or writing.

This function returns a `DmOpenRef` which must be used to access particular records in a database. If unsuccessful, 0 is returned and the cause of the error can be determined by calling `DmGetLastErr()`.

When you use this function to open a resource database in read-only mode, it also opens the overlay associated with this database for the current locale, if it exists. (The function `DmGetOverlayLocale()` returns the current locale.) Overlays are resource databases typically used to localize applications, shared libraries, and panels. They have the same creator as the base database, a type of `'ovly'` (symbolically named `omOverlayDBType`), and contain resources with the same IDs and types as the resources in the base database. When you request a resource from the database using `DmGetResource()` or `DmGet1ResourceV50()`, the overlay is searched first. If the overlay contains a resource for the given ID, it is returned. If not, the resource from the base database is returned.

The `DmOpenRef` returned by this function is the pointer to the base database, not to the overlay database, so care should be taken when passing this pointer to functions such as `DmFindResource()` because this circumvents the overlay.

It's possible to create a "stripped" base resource database, one that does not contain any user interface resources. `DmOpenDatabase()` only opens a stripped database if its corresponding overlay exists. If the overlay does not exist or if the overlay doesn't match the resource database, `DmOpenDatabase()` returns `NULL` and `DmGetLastErr()` returns the error code `omErrBaseRequiresOverlay`.

If you open a resource database in a writable mode, the associated overlay is not opened. If you make changes to the resource database, the overlay database is invalidated if those changes affect any resources that are also in the overlay. This means that on future occasions where you open the resource database in read-only mode, the overlay will not be opened because Palm OS considers it to be invalid.

IMPORTANT: When called from the main application thread, this function may block. While blocked, the application will not receive events and won't redraw its windows. As well, deferred sublaunches and notifications won't execute while the main application thread is blocked.

See Also `DbOpenDatabase()`, `DmCloseDatabase()`, `DmCreateDatabase()`, `DmFindDatabase()`, `DmOpenDatabaseByTypeCreator()`, `DmDeleteDatabase()`, `DmOpenDBNoOverlay()`

DmOpenDatabaseByTypeCreator Function

Purpose Open the most recent revision of a database with the given type and creator. If the database is a resource database, also open its overlay for the current locale.

Declared In `DataMgr.h`

Prototype

```
DmOpenRef DmOpenDatabaseByTypeCreator
   (uint32_t type, uint32_t creator,
   DmOpenModeType mode)
```

Parameters

→ *`type`*
Type of database.

→ *`creator`*
Creator of database.

→ *`mode`*
Which mode to open database in (see `DmOpenModeType`).

Returns `DmOpenRef` to open database. Unlike `DmOpenDatabaseByTypeCreatorV50()`, no fatal error message is displayed; if the database couldn't be found this function simply returns 0 and `DmGetLastErr()` returns an error code indicating the reason for failure.

Comments If you use this function to open a resource database in read-only mode, it also opens the overlay associated with this database for the current locale. See `DmOpenDatabase()` for more information on overlays and resource databases.

IMPORTANT: When called from the main application thread, this function may block. While blocked, the application will not receive events and won't redraw its windows. As well, deferred sublaunches and notifications won't execute while the main application thread is blocked.

See Also `DmFindDatabaseByTypeCreator()`, `DmOpenDatabase()DmOpenDBNoOverlay()DmOpenIteratorByTypeCreator()`

DmOpenDatabaseByTypeCreatorV50 Function

Purpose Opens the most recent revision of a classic database or extended resource database with the given type and creator. If the database is a resource database, either classic or extended, this function also opens its overlay for the current locale.

Declared In `DataMgr.h`

Prototype
```
DmOpenRef DmOpenDatabaseByTypeCreatorV50
    (uint32_t type, uint32_t creator,
    DmOpenModeType mode)
```

Parameters

→ *`type`*
Type of database.

→ *`creator`*
Creator of database.

→ *`mode`*
Which mode to open database in (see `DmOpenModeType`).

Returns `DmOpenRef` to open database. If the database couldn't be found this function returns 0 and `DmGetLastErr()` returns an error code indicating the reason for failure.

Comments If you use this function to open a resource database in read-only mode, it also opens the overlay associated with this database for the current locale. See `DmOpenDatabase()` for more information on overlays and resource databases.

IMPORTANT: When called from the main application thread, this function may block. While blocked, the application will not receive events and won't redraw its windows. As well, deferred sublaunches and notifications won't execute while the main application thread is blocked.

Compatibility This function operates only on classic databases, and exists for compatibility purposes only. Palm OS Cobalt applications should use `DmOpenDatabaseByTypeCreator()` instead.

See Also `DmOpenDatabaseByTypeCreator()`, `DmCreateDatabase()`, `DmOpenDatabase()`, `DmOpenDatabaseInfoV50()`, `DmCloseDatabase()`, `DmOpenDBNoOverlay()`

DmOpenDatabaseInfoV50 Function

Purpose Retrieve information about an open database.

Declared In `DataMgr.h`

Prototype

```
status_t DmOpenDatabaseInfoV50 (DmOpenRef dbRef,
    LocalID *pDbID, uint16_t *pOpenCount,
    DmOpenModeType *pMode, uint16_t *pCardNo,
    Boolean *pResDB)
```

Parameters

→ *dbRef*
: `DmOpenRef` to an open database.

← *pDbID*
: The ID of the database. Pass `NULL` for this parameter if you don't want to retrieve this information.

← *pOpenCount*
: The number of applications that have this database open. Pass `NULL` for this parameter if you don't want to retrieve this information.

← *pMode*
: The mode used to open the database (see `DmOpenModeType`). Pass `NULL` for this parameter if you don't want to retrieve this information.

← *pCardNo*
: The number of the card on which this database resides. Pass `NULL` for this parameter if you don't want to retrieve this information.

← *pResDB*
: If `true` upon return, the database is a resource database, `false` otherwise. Pass `NULL` for this parameter if you don't want to retrieve this information.

Returns Returns `errNone` if no error.

Compatibility This function is provided only to ease the porting of applications from previous versions of Palm OS. Palm OS Cobalt applications will want to use `DmGetOpenInfo()` instead.

See Also `DmDatabaseInfo()`

DmOpenDatabaseV50 Function

Purpose Open a non-schema database and return a reference to it. If the database is a resource database, also open its overlay for the current locale.

Declared In `DataMgr.h`

Prototype
```
DmOpenRef DmOpenDatabaseV50 (uint16_t cardNo,
   LocalID dbID, DmOpenModeType mode)
```

Parameters

→ *cardNo*
Card number database resides on.

→ *dbID*
The database ID of the database.

→ *mode*
Which mode to open database in (see `DmOpenModeType`).

Returns Returns `DmOpenRef` to open database. May display a fatal error message if the database parameter is `NULL`. On all other errors, this function returns 0 and `DmGetLastErr()` returns an error code indicating the reason for failure.

Comments Call this function to open a database for reading or writing.

This function returns a `DmOpenRef` which must be used to access particular records in a database. If unsuccessful, 0 is returned and the cause of the error can be determined by calling `DmGetLastErr()`.

When you use this function to open a resource database in read-only mode, it also opens the overlay associated with this database for the current locale, if it exists. (The function `DmGetOverlayLocale()` returns the current locale.) Overlays are resource databases typically used to localize applications, shared libraries, and panels. They have the same creator as the base database, a type of `'ovly'` (symbolically named `omOverlayDBType`), and contain resources with the same IDs and types as the resources in the base database. When you request a resource from the database using `DmGetResource()` or `DmGet1ResourceV50()`, the overlay is searched first. If the overlay contains a resource for the given ID, it is returned. If not, the resource from the base database is returned.

The `DmOpenRef` returned by this function is the pointer to the base database, not to the overlay database, so care should be taken when passing this pointer to functions such as `DmFindResource()` because this circumvents the overlay.

It's possible to create a "stripped" base resource database, one that does not contain any user interface resources. `DmOpenDatabaseV50()` only opens a stripped database if its corresponding overlay exists. If the overlay does not exist or if the overlay doesn't match the resource database, `DmOpenDatabaseV50()` returns `NULL` and `DmGetLastErr()` returns the error code `omErrBaseRequiresOverlay`.

If you open a resource database in a writable mode, the associated overlay is not opened. If you make changes to the resource database, the overlay database is invalidated if those changes affect any resources that are also in the overlay. This means that on future occasions where you open the resource database in read-only mode, the overlay will not be opened because Palm OS considers it to be invalid.

TIP: If you want to prevent your resource database from being overlaid, include an `'xprf'` resource (symbolically named `sysResTExtPrefs`) in the database with the ID 0 (`sysResIDExtPrefs`) and set its `disableOverlays` flag. This resource is defined in `UIResources.r`.

When `DmOpenDatabaseV50()` attempts to open a stripped resource database and cannot find an overlay for it, it searches for an overlay matching the default locale if the system locale is different from the default locale.

IMPORTANT: When called from the main application thread, this function may block. While blocked, the application will not receive events and won't redraw its windows. As well, deferred sublaunches and notifications won't execute while the main application thread is blocked.

Compatibility This function is provided only to ease the porting of applications from previous versions of Palm OS. Palm OS Cobalt applications will want to use `DmOpenDatabase()` instead.

See Also `DmOpenDatabase()`, `DmCloseDatabase()`, `DmCreateDatabase()`, `DmFindDatabase()`, `DmOpenDatabaseByTypeCreator()`, `DmDeleteDatabase()`, `DmOpenDBNoOverlay()`

DmOpenDBNoOverlay Function

Purpose Open a non-schema database and return a reference to it.

Declared In `DataMgr.h`

Prototype
```
DmOpenRef DmOpenDBNoOverlay (DatabaseID dbID,
   DmOpenModeType mode)
```

Parameters

→ *dbID*
Database ID of the database.

→ *mode*
Which mode to open database in (see `DmOpenModeType`).

Returns Returns a `DmOpenRef` to the open database. Unlike `DmOpenDBNoOverlayV50()`, no fatal error message is displayed; on error, this function simply returns 0 and `DmGetLastErr()` returns an error code indicating the reason for failure.

Comments Call this function to open a database for reading or writing, while ignoring any overlay databases that might be associated with it.

This function returns a `DmOpenRef` which must be used to access particular records in a database. If unsuccessful, 0 is returned and the cause of the error can be determined by calling `DmGetLastErr()`.

IMPORTANT: When called from the main application thread, this function may block. While blocked, the application will not receive events and won't redraw its windows. As well, deferred sublaunches and notifications won't execute while the main application thread is blocked.

See Also `DmCloseDatabase()`, `DmCreateDatabase()`, `DmFindDatabase()`, `DmOpenDatabaseByTypeCreator()`, `DmDeleteDatabase()`, `DmOpenDatabase()`

DmOpenDBNoOverlayV50 Function

Purpose Open a non-schema database and return a reference to it.

Declared In `DataMgr.h`

Prototype

```
DmOpenRef DmOpenDBNoOverlayV50 (uint16_t cardNo,
    LocalID dbID, DmOpenModeType mode)
```

Parameters

→ *`cardNo`*
Card number database resides on.

→ *`dbID`*
The database ID of the database.

→ *`mode`*
Which mode to open database in (see `DmOpenModeType`).

Returns `DmOpenRef` to open database. May display a fatal error message if the database parameter is `NULL`. On all other errors, this function returns 0 and `DmGetLastErr()` returns an error code indicating the reason for failure.

Comments Call this function to open a database for reading or writing, while ignoring any overlay databases that might be associated with it.

This function returns a `DmOpenRef` which must be used to access particular records in a database. If unsuccessful, 0 is returned and the cause of the error can be determined by calling `DmGetLastErr()`.

IMPORTANT: When called from the main application thread, this function may block. While blocked, the application will not receive events and won't redraw its windows. As well, deferred sublaunches and notifications won't execute while the main application thread is blocked.

Compatibility This function is provided only to ease the porting of applications from previous versions of Palm OS. Palm OS Cobalt applications will want to use `DmOpenDBNoOverlay()` instead.

See Also `DmOpenDBNoOverlay()`, `DmCloseDatabase()`, `DmCreateDatabase()`, `DmFindDatabase()`, `DmOpenDatabaseByTypeCreator()`, `DmDeleteDatabase()`, `DmOpenDatabase()`

DmOpenIteratorByTypeCreator Function

Purpose Mark the start of an iteration through those databases that match a specified set of criteria.

Declared In `DataMgr.h`

Prototype

```
status_t DmOpenIteratorByTypeCreator
    (DmSearchStatePtr stateInfoP, uint32_t type,
    uint32_t creator, Boolean onlyLatestVers,
    DmFindType find)
```

Parameters

→ *`stateInfoP`*
Pointer to a `DmSearchStateType` structure that you have allocated. The iteration process uses this opaque structure to maintain its state.

→ *`type`*
Type of database to search for, pass `dmSearchWildcardID` to iterate through databases of all types.

→ *`creator`*
Creator of database to search for, pass `dmSearchWildcardID` to iterate through databases with all creator IDs.

→ *`onlyLatestVers`*
If `true`, only the latest version of a database with a given type and creator is returned.

→ *`find`*
Flags indicating the type of database to be searched for: schema, extended, classic, or a combination of the three. See `DmFindType` for more information.

Returns Returns `errNone`.

Comments See the comments under `DmGetNextDatabaseByTypeCreator()` for an example of how this function is used.

See Also `DmGetNextDatabaseByTypeCreator()`, `DmCloseIteratorByTypeCreator()`

DmPtrResize Function

Purpose Resize a storage heap chunk given a pointer to its data.

Declared In `DataMgr.h`

Prototype `status_t DmPtrResize (MemPtr` *`p`*`, uint32_t` *`newSize`*`)`

Parameters → *`p`*
Pointer to the chunk.

→ *`newSize`*
The new desired size.

Returns Returns `errNone` if the chunk was successfully resized, or one of the following if an error occurred:

`dmErrInvalidParam`
The function received an invalid parameter.

`memErrNotEnoughSpace`
A memory error occurred.

`memErrChunkLocked`
The associated memory chunk is locked.

Comments Call this function to resize a locked chunk. This function is always successful when shrinking the size of a chunk. When growing a chunk, it attempts to use free space immediately following the chunk.

See Also `DmPtrSize()`, `DmHandleResize()`

DmPtrSize Function

Purpose Return the size of a storage heap chunk given a pointer to its data.

Declared In `DataMgr.h`

Prototype `uint32_t DmPtrSize (MemPtr p)`

Parameters → *p*

Pointer to the chunk.

Returns The requested size of the chunk.

Comments Call this function to get the original requested size of a chunk.

DmPtrUnlock Function

Purpose Unlock a storage heap chunk, given a pointer to its data.

Declared In `DataMgr.h`

Prototype `status_t DmPtrUnlock (MemPtr p)`

Parameters → *p*

Pointer to a chunk.

Returns Returns `errNone` if the chunk was successfully unlocked, or `dmErrInvalidParam` if there was a problem with the chunk pointer.

Comments A chunk must *not* be unlocked more times than it was locked.

See Also `DmHandleLock()`

DmQueryNextInCategory Function

Purpose Return a handle to the next record in the specified category for reading only (does not set the `busy` bit).

Declared In `DataMgr.h`

Prototype
```
MemHandle DmQueryNextInCategory (DmOpenRef dbRef,
   uint16_t *pIndex, uint16_t category)
```

Parameters

→ *`dbRef`*
`DmOpenRef` to an open database.

↔ *`pIndex`*
Index of a known record (often retrieved with `DmGetPositionInCategory()`). If a "next" record is found, this index is updated to indicate that record.

→ *`category`*
Index of category to query, or `dmAllCategories` to find the next record in any category.

Returns Returns a handle to the record, along with the index of that record. If a record couldn't be found, this function returns `NULL`, and `DmGetLastErr()` returns an error code indicating the reason for failure.

Comments This function begins searching the database from the record at *`*pIndex`* for a record that is in the specified category. If the record at *`*pIndex`* belongs to that category, then a handle to it is returned. If not, the function continues searching until it finds a record in the category.

Records that have the `deleted` bit set are skipped, and if the user has specified that private records should be hidden or masked, private records are skipped as well.

Because this function begins searching the database at the record with the supplied index, if you want to find the next record in the category after the one you have an index for, increment the index value before calling this function. For example:

```
DmOpenRef myDB;   //assume that this is set
uint16_t recIndex;   //assume that this is set
uint8_t category;
status_t err;
uint16_t pos;
MemHandle newRecH;
```

```
err = DmGetRecordCategory(myDb, recIndex, &category);
pos = DmGetPositionInCategory(myDB, recIndex, category);
pos++;   //advance to next record
newRecH = DmQueryNextInCategory(myDB, &pos, category);
```

See Also `DmNumRecordsInCategory()`, `DmGetPositionInCategory()`, `DmFindRecordByOffsetInCategory()`

DmQueryRecord Function

Purpose Return a handle to a record for reading only (does not set the `busy` bit).

Declared In `DataMgr.h`

Prototype
```
MemHandle DmQueryRecord (DmOpenRef dbRef,
    uint16_t index)
```

Parameters

→ *`dbRef`*
`DmOpenRef` to an open database.

→ *`index`*
Which record to retrieve.

Returns Returns a record handle. If an error occurs, this function returns `NULL`, and `DmGetLastErr()` returns an error code indicating the reason for failure.

Some releases may display a fatal error message if the specified index is out of range.

Comments Returns a handle to the given record. Use this function only when viewing the record. This function successfully returns a handle to the record even if the record is busy.

If the record is ROM-based (pointer accessed) this function returns the fake handle to it.

DmQuickSort Function

Purpose Sort records in a database.

Declared In `DataMgr.h`

Prototype
```
status_t DmQuickSort (const DmOpenRef dbR,
   DmCompareFunctionType *compar, int16_t other)
```

Parameters

→ *`dbR`*
Database access pointer.

→ *`compar`*
Comparison function. See `DmCompareFunctionType()`.

→ *`other`*
Any value the application wants to pass to the comparison function. This parameter is often used to indicate a sort direction (ascending or descending).

Returns Returns `errNone` if no error, or one of the following if an error occurs:

`dmErrReadOnly`
You've attempted to write to or modify a database that is open in read-only mode.

`dmErrNotRecordDB`
You've attempted to perform a record function on a resource database.

Some releases may display a fatal error message instead of returning the error code.

Comments Deleted records are placed last in any order. All others are sorted according to the passed comparison function.

After `DmQuickSort()` returns, equal database records do not have a consistent order. That is, if `DmQuickSort()` is passed two equal records, their resulting order is unpredictable. To prevent records that contain the same data from being rearranged in an unpredictable order, pass the record's unique ID to the comparison function (using the `DmSortRecordInfoType` structure).

`DmQuickSort()` contains its own stack to limit uncontrolled recursion. When the stack is full `DmQuickSort()` instead performs an insertion sort. An insertion sort is also performed when the number of records is low, avoiding the noticeable overhead of a quick sort with a small number of records. Finally, if the records seem mostly sorted an insertion sort is performed to move only those records that need moving.

See Also `DmInsertionSort()`

DmRecordInfoV50 Function

Purpose Retrieve the record information stored in the database header.

Declared In `DataMgr.h`

Prototype

```
status_t DmRecordInfoV50 (DmOpenRef dbRef,
   uint16_t index, uint16_t *pAttr,
   uint32_t *pUID, LocalID *pChunkID)
```

Parameters

→ *`dbRef`*
`DmOpenRef` to an open database.

→ *`index`*
Index of the record.

← *`pAttr`*
The record's attributes. See "Non-Schema Database Record Attributes." Pass `NULL` for this parameter if you don't want to retrieve this value.

← *`pUID`*
The record's unique ID. Pass `NULL` for this parameter if you don't want to retrieve this value.

← *`pChunkID`*
The record's local ID. Pass `NULL` for this parameter if you don't want to retrieve this value.

Returns Returns `errNone` if no error or `dmErrIndexOutOfRange` if the specified record can't be found. Some releases may display a fatal error message instead of returning the error code.

Compatibility This function is provided for compatibility purposes only. Palm OS Cobalt applications should use one or more of the functions listed in the See Also section, below, instead.

See Also `DmGetRecordAttr()`, `DmGetRecordCategory()`, `DmGetRecordID()`, `DmQueryNextInCategory()`

DmRecoverHandle Function

Purpose Recover the handle of a storage heap chunk, given a pointer to its data.

Declared In `DataMgr.h`

Prototype `MemHandle DmRecoverHandle (MemPtr pChunk)`

Parameters → *`pChunk`*
Pointer to the chunk.

Returns Returns the handle of the chunk, or 0 if unsuccessful.

Comments Don't call this function for pointers in ROM.

DmReleaseRecord Function

Purpose Clear the `busy` bit for the given record and set the `dirty` bit if *`fDirty`* is `true`.

Declared In `DataMgr.h`

Prototype
```
status_t DmReleaseRecord (DmOpenRef dbRef,
   uint16_t index, Boolean fDirty)
```

Parameters → *`dbRef`*
`DmOpenRef` to an open database.

→ *`index`*
The record to unlock.

→ *`fDirty`*
If `true`, set the `dirty` bit.

Returns Returns `errNone` if no error, or `dmErrIndexOutOfRange` if the specified index is out of range. Some releases may display a fatal error message instead of returning the error code.

Comments Call this function when you finish modifying or reading a record that you've called `DmGetRecord()` on or created using `DmNewRecord()`.

See Also `DmGetRecord()`

DmReleaseResource Function

Purpose Release a resource acquired with `DmGetResource()`.

Declared In `DataMgr.h`

Prototype
```
status_t DmReleaseResource (MemHandle hResource)
```

Parameters
→ *hResource*
Handle to resource.

Returns Returns `errNone` if no error.

Comments Marks a resource as being no longer needed by the application.

See Also DmGet1ResourceV50(), DmGetResource()

DmRemoveRecord Function

Purpose Remove a record from a database and dispose of its data chunk.

Declared In `DataMgr.h`

Prototype
```
status_t DmRemoveRecord (DmOpenRef dbRef,
   uint16_t index)
```

Parameters
→ *dbRef*
`DmOpenRef` to an open database.

→ *index*
Index of the record to remove.

Returns Returns `errNone` if no error, or one of the following if an error occurs:

`dmErrReadOnly`
You've attempted to write to or modify a database that is open in read-only mode.

`dmErrIndexOutOfRange`
The specified index is out of range.

`dmErrNotRecordDB`
You've attempted to perform a record function on a resource database.

`memErrChunkLocked`
The associated memory chunk is locked.

`memErrInvalidParam`
  A memory error occurred.

Some releases may display a fatal error message instead of returning the error code.

Comments Disposes of the record's data chunk and removes the record's entry from the database header. `DmRemoveRecord()` should only be used for newly-created records that have just been deleted or records that have never been synchronized.

See Also DmDetachRecord(), DmDeleteRecord(), DmArchiveRecord(), DmNewRecord()

DmRemoveResource Function

Purpose Delete a resource from a resource database.

Declared In `DataMgr.h`

Prototype
```
status_t DmRemoveResource (DmOpenRef dbRef,
   uint16_t index)
```

Parameters

→ *`dbRef`*
  `DmOpenRef` to an open database.

→ *`index`*
  Index of resource to delete.

Returns Returns `errNone` if no error, or one of the following if an error occurs:

`dmErrCorruptDatabase`
  The database is corrupted.

`dmErrIndexOutOfRange`
  The specified index is out of range.

`dmErrReadOnly`
  You've attempted to write to or modify a database that is open in read-only mode.

`memErrChunkLocked`
  The associated memory chunk is locked.

`memErrInvalidParam`
  A memory error occurred.

`memErrNotEnoughSpace`
A memory error occurred.

May display a fatal error message if the database is not a resource database.

Comments This function disposes of the Memory Manager chunk that holds the given resource and removes its entry from the database header.

See Also DmDetachResource(), DmRemoveResource(), DmAttachResource()

DmRemoveSecretRecords Function

Purpose Remove all secret records.

Declared In `DataMgr.h`

Prototype `status_t DmRemoveSecretRecords (DmOpenRef` *`dbRef`*`)`

Parameters → *`dbRef`*
`DmOpenRef` to an open database.

Returns Returns `errNone` if no error, or one of the following if an error occurs:

`dmErrReadOnly`
You've attempted to write to or modify a database that is open in read-only mode.

`dmErrNotRecordDB`
You've attempted to perform a record function on a resource database.

Some releases may display a fatal error message instead of returning the error code.

See Also DmRemoveRecord(), DmRecordInfoV50(), DmSetRecordInfoV50()

DmResetRecordStates Function

Purpose For each record in a non-schema database, unlocks the record and clears the busy bit.

Declared In `DataMgr.h`

Prototype `status_t DmResetRecordStates (DmOpenRef dbRef)`

Parameters → *`dbRef`*
`DmOpenRef` to an open non-schema database.

Returns Returns `errNone` if the operation completed successfully, or one of the following otherwise:

`dmErrInvalidParam`
`dbRef` doesn't reference an open database, or *`dbRef`* references a schema database.

`dmErrReadOnly`
The specified database isn't open for writing.

`dmErrROMBased`
The specified database is located in ROM.

`memErrInvalidParam`
A memory error occurred.

See Also `DmSetRecordAttr()`

DmResizeRecord Function

Purpose Resize a record by index.

Declared In `DataMgr.h`

Prototype
```
MemHandle DmResizeRecord (DmOpenRef dbRef,
   uint16_t index, uint32_t newSize)
```

Parameters → *`dbRef`*
`DmOpenRef` to an open database.

→ *`index`*
Which record to retrieve.

→ *`newSize`*
New size of record.

Returns Handle to resized record. Returns `NULL` if there is not enough space to resize the record, and `DmGetLastErr()` returns an error code indicating the reason for failure. Some releases may display a fatal error message instead of returning the error code.

Comments As this function reallocates the record, the handle may change, so be sure to use the returned handle to access the resized record.

DmResizeResource Function

Purpose Resize a resource and return the new handle.

Declared In `DataMgr.h`

Prototype

```
MemHandle DmResizeResource (MemHandle hResource,
   uint32_t size)
```

Parameters

→ *`hResource`*
Handle to resource.

→ *`size`*
Desired new size of resource.

Returns Returns a handle to newly sized resource. Returns `NULL` if there is not enough space to resize the resource, and `DmGetLastErr()` returns an error code indicating the reason for failure. Some releases may display a fatal error message instead of returning the error code.

Comments Resizes the resource and returns a new handle.

The handle may change if the resource had to be reallocated in a different data heap because there was not enough space in its present data heap.

DmResourceInfo Function

Purpose Retrieve information on a given resource.

Declared In `DataMgr.h`

Prototype

```
status_t DmResourceInfo (DmOpenRef dbRef,
   uint16_t index, DmResourceType *pResType,
   DmResourceID *pResID, MemHandle *pChunkHandle)
```

Parameters

→ *`dbRef`*
`DmOpenRef` to an open database.

→ *`index`*
Index of resource to get info on.

← *`pResType`*
The resource type. Pass `NULL` if you don't want to retrieve this information.

← *`pResID`*
The resource ID. Pass `NULL` if you don't want to retrieve this information.

← *`pChunkHandle`*
Handle for the resource data. Pass `NULL` if you don't want to retrieve this information.

Returns Returns `errNone` if no error or `dmErrIndexOutOfRange` if an error occurred. Unlike `DmResourceInfoV50()`, no fatal error message is displayed if the database is not a resource database.

Comments If *`dbRef`* is a pointer to a base resource database, the information returned is about the resource from that database alone; this function ignores any associated overlay.

See Also DmGetResource(), DmGet1ResourceV50(), DmSetResourceInfo(), DmFindResource(), DmFindResourceType()

DmResourceInfoV50 Function

Purpose Retrieve information on a given resource.

Declared In `DataMgr.h`

Prototype

```
status_t DmResourceInfoV50 (DmOpenRef dbRef,
   uint16_t index, DmResourceType *pResType,
   DmResourceID *pResID, LocalID *pChunkLocalID)
```

Parameters

→ *dbRef*
: `DmOpenRef` to an open database.

→ *index*
: Index of resource to get info on.

← *pResType*
: The resource type. Pass `NULL` if you don't want to retrieve this information.

← *pResID*
: The resource ID. Pass `NULL` if you don't want to retrieve this information.

← *pChunkLocalID*
: The Memory Manager local ID of the resource data. Pass `NULL` if you don't want to retrieve this information.

Returns Returns `errNone` if no error or `dmErrIndexOutOfRange` if an error occurred. May display a fatal error message if the database is not a resource database.

Comments If *dbRef* is a pointer to a base resource database, the information returned is about the resource from that database alone; this function ignores any associated overlay.

Compatibility This function is provided for compatibility purposes only. Palm OS Cobalt applications should use `DmResourceInfo()` instead.

See Also DmResourceInfo(), DmGetResource(), DmGet1ResourceV50(), DmSetResourceInfo(), DmFindResource(), DmFindResourceType()

DmRestoreFinalize Function

Purpose Complete or abort an on-going database restore operation.

Declared In `DataMgr.h`

Prototype

```
status_t DmRestoreFinalize
   (DmBackupRestoreStatePtr pState,
   Boolean fAbort, Boolean fOverwrite,
   DatabaseID *pDbID)
```

Parameters

→ *`pState`*
Pointer to a `DmBackupRestoreStateType` structure allocated by the caller and initialized with `DmBackupInitialize()`.

→ *`fAbort`*
Set to `true` to abort an on-going backup operation, or `false` to clean up after a successful backup.

→ *`fOverwrite`*
Set to `true` to overwrite an existing matching database (if there is one), or `false` to leave the existing matching database intact.

← *`pDbID`*
Pointer to a variable that receives the identifier for the restored database, or `NULL` if the database identifier isn't needed.

Returns Returns `errNone` if the database image was successfully restored, `dmErrOperationAborted` if the restore operation was cancelled, or one of the following errors otherwise:

`dmErrInvalidParam`
One of the parameters is invalid or corrupt.

`dmErrMemError`
A memory error occurred.

`dmErrAlreadyExists`
The database being restored already exists, and the *`fOverwrite`* parameter was set to `false`.

Comments This function allows the Data Manager to perform a final clean up of the internal structures it allocated for the operation. Applications should always call this function after having started a restore operation, whether or not the restore completed successfully. See

`DmRestoreUpdate()` for sample code illustrating this function's use.

The restore operation can be used with schema, extended, or classic databases.

See Also `DmBackupFinalize()`, `DmRestoreInitialize()`

DmRestoreInitialize Function

Purpose Initialize the Data Manager prior to starting a restore operation on the specified database.

Declared In `DataMgr.h`

Prototype

```
status_t DmRestoreInitialize
    (DmBackupRestoreStatePtr pState,
    DmDatabaseInfoPtr pDbInfo)
```

Parameters

→ *pState*
: Pointer to a `DmBackupRestoreStateType` structure allocated by the caller.

→ *pDbInfo*
: Pointer to a `DmDatabaseInfoType` structure that will receive information about the database being restored. This structure will receive its information after you call `DmRestoreUpdate()`. Set to `NULL` if you don't want to receive this information.

Returns Returns `errNone` if the initialization was successful, or one of the following if an error occurred:

`dmErrAccessDenied`
: The caller was not authorized to perform a restore operation for the specified database.

`dmErrInvalidParam`
: One of the parameters is invalid or corrupt.

`dmErrMemError`
: A memory error occurred.

Comments Use `DmRestoreInitialize()` to start a database backup operation. See `DmRestoreUpdate()` for sample code illustrating this function's use.

The restore operation can be used with schema, extended, or classic databases.

IMPORTANT: When called from the main application thread, this function may block. While blocked, the application will not receive events and won't redraw its windows. As well, deferred sublaunches and notifications won't execute while the main application thread is blocked.

See Also `DmBackupInitialize()`, `DmRestoreFinalize()`

DmRestoreUpdate Function

Purpose Reassemble a database within the storage heap from a database image stream held within the specified buffer.

Declared In `DataMgr.h`

Prototype

```
status_t DmRestoreUpdate
   (DmBackupRestoreStatePtr pState,
   MemPtr pBuffer, uint32_t size,
   Boolean endOfData, Boolean *pfDbInfoAvailable)
```

Parameters

→ *`pState`*
Pointer to a `DmBackupRestoreStateType` structure allocated by the caller and initialized with `DmRestoreInitialize()`.

→ *`pBuffer`*
Pointer to a buffer to hold the backed-up database image that is being restored.

→ *`size`*
Size, in bytes, of the database image data held within *`pBuffer`*.

→ *`endOfData`*
Set this parameter to `true` to indicate that there is no additional data (beyond what is in *`pBuffer`*). Set it to `false` if you will be making additional calls to `DmRestoreUpdate()`.

← *`pfDbInfoAvailable`*
Pointer to a Boolean variable that is to indicate whether the information about the database being restored is available, or `NULL` if you don't need the database information. If true, the information was written to the `DmDatabaseInfoType` structure you specified when calling `DmRestoreInitialize()`.

Returns Returns `errNone` if the operation was successful, or one of the following if an error occurred:

`dmErrInvalidParam`
One of the parameters is invalid or corrupt.

`dmErrMemError`
A memory error occurred which prevented the restore operation from continuing.

Comments Use this function, along with `DmRestoreInitialize()` and `DmRestoreFinalize()`, to restore a schema, extended, or classic database from its serial image.

If the serial image doesn't reside in a single buffer, you'll need to call this function several times before you've completely restored the complete database. Call `DmRestoreUpdate()` as many times as required until all of the database image data has been successfully processed by this function. For all but the last call to this function, *`endOfData`* must be set to `false`. The last time you call it, set *`endOfData`* to `true` (note that the last call needn't contain any data in *`pBuffer`*; see the example, below, for code that does this). Finally, call `DmRestoreFinalize()` to complete the operation and have the database once again accessible from the Data Manager's database directory list.

If *`pfDbInfoAvailable`* is not `NULL`, `DmRestoreUpdate()` sets the pointed-to Boolean variable to `true` when it has received enough of the database image to be able to return information about it. The actual database information is returned through the `DmDatabaseInfoType` structure that you specified when calling `DmRestoreInitialize()`.

If `DmRestoreUpdate()` returns an error code other than `errNone`, the operation has been aborted due to a fatal error. You must still perform a call to `DmRestoreFinalize()` to let the Data Manager perform a final cleanup of the internal structures it allocated for the operation.

Example This sample code shows how to use `DmRestoreInitialize()`, `DmRestoreUpdate()`, and `DmRestoreFinalize()` to restore database from a serial image. This code employs a fictitious `DoesUserWantToOverwrite()` function to let the user decide whether to overwrite a matching database (if any).

```
status_t error;
DatabaseID dbID;
DmBackupRestoreStateType restoreState;
char buffer[BUFFER_SIZE];
uint32_t size;
Boolean fAbort;
Boolean fGotDbInfo;
Boolean fDone = false;
Boolean fOverwrite = false;
Boolean fAlreadyAsked = false;
DmDatabaseInfoType databaseInfo;
char dbName[dmDBNameLength];
uint32_t type;
uint32_t creator;
uint16_t attributes;

// Set up the DmDatabaseInfoType structure so that we will
// get the information we want about the database being
// restored...
MemSet(&databaseInfo, sizeof(databaseInfo), 0);
databaseInfo.pName = dbName;
databaseInfo.pType = &type;
databaseInfo.pCreator = &creator;
databaseInfo.pAttributes = &attributes;

error = DmRestoreInitialize(&restoreState, &databaseInfo);
if (error == errNone) {
   do {
      size = sizeof(buffer);

      // Get a chunk from the database image data out of some
      // I/O channel. We assume this function returns false
      // when there is no more data to receive for the
      // database image.
      if (GetDatabaseImageData(buffer, &size)) {
         error = DmRestoreUpdate(&restoreState, buffer,
            size, false, &fGotDbInfo);
         // Set the abort flag if we got back an error or if
         // the user decided to cancel the operation...
         fAbort = (error != errNone) | DidUserCancel();

         if (!fAbort && fGotDbInfo && !fAlreadyAsked) {
            // We just got the database info we asked so now
            // we ask the user whether they want to
            // overwrite the existing database with this
            // one...
            fOverwrite = DoesUserWantToOverwrite(&pDbInfo,
               &fFoundDb);

            // If the user doesn't want to overwrite and we
            // found an existing database in the storage
            // heap, then set the abort flag to break out of
            // the loop.
            fAbort = !fOverwrite && fFoundDb;

            // Use this flag to make sure we don't ask the
            // user twice (or more) the same question in case
            // where we didn't find a matching database or
            // they wanted to overwrite anyway...
            fAlreadyAsked = true;
         }
      } else
         fDone = true;

   } while(!fDone && !fAbort);

   // call DmRestoreUpdate one last time with no data and
   // with the endOfData flag set to mark the end of data
   error = DmRestoreUpdate(&restoreState, buffer,
      size, true, &fGotDbInfo);

   // Always call DmRestoreFinalize to complete the restore
   // operation ...
   error = DmRestoreFinalize(&restoreState, fAbort,
      fOverwrite, &dbID);
}

if (error == errNone) {
   // Restore operation completed successfully...

   // Now we can use the dbID we got back to operate on the
   // newly-restored database. Note also that we can also use
   // the database information we got back during the restore
   // operation.
} else {
   // A fatal error occurred...

   if (error == dmErrOperationAborted) {
      // The user aborted. Handle it.
   } else
      if (error == dmErrAlreadyExists) {
         // The database already exists! Handle this.
      } else {
         // Some other error occurred.
   }
}
```

See Also `DmBackupUpdate()`, `DmCreateDatabaseFromImage()`

DmSearchRecordOpenDatabases Function

Purpose Search all open record databases for a record with the handle passed.

Declared In `DataMgr.h`

Prototype
```
uint16_t DmSearchRecordOpenDatabases
   (MemHandle hRecord, DmOpenRef *pDbRef)
```

Parameters

→ *`hRecord`*
Record handle.

← *`pDbRef`*
The database that contains the record *`hRecord`*.

Returns Returns the index of the record and database access pointer; if not found, returns -1 and `*`*`pDbRef`* is 0.

See Also DmGetRecord(), DmFindRecordByID(), DmRecordInfoV50()

DmSearchResourceOpenDatabases Function

Purpose Search all open resource databases for a resource by type and ID, or by pointer if it is non-`NULL`.

Declared In `DataMgr.h`

Prototype
```
uint16_t DmSearchResourceOpenDatabases
    (DmResourceType resType, DmResourceID resID,
    MemHandle hResource, DmOpenRef *pDbRef)
```

Parameters

→ *`resType`*
Type of resource to search for.

→ *`resID`*
ID of resource to search for.

→ *`hResource`*
Handle of locked resource, or `NULL`.

← *`pDbRef`*
The resource database that contains the specified resource.

Returns Returns the index of the resource, stores `DmOpenRef` in `*`*`pDbRef`*.

Comments This function can be used to find a resource in all open resource databases by type and ID or by pointer. If *`hResource`* is `NULL`, the resource is searched for by type and ID. If *`hResource`* is not `NULL`, *`resType`* and *`resID`* is ignored and the index of the resource handle is returned. On return, `*`*`pDbRef`* contains the access pointer of the resource database that the resource was eventually found in. Once the index of a resource is determined, it can be locked down and accessed by calling `DmGetResourceByIndex()`.

If any of the open databases are overlaid, this function finds and returns the localized version of the resource when searching by type and creator. In this case, the *`pDbRef`* return value is a pointer to the overlay database, not the base resource database.

See Also DmGetResource(), DmFindResourceType(), DmResourceInfo(), DmFindResource()

DmSet Function

Purpose Write a specified value into a section of a record. This function also checks the validity of the pointer for the record and makes sure the writing of the record information doesn't exceed the bounds of the record.

Declared In `DataMgr.h`

Prototype

```
status_t DmSet (void *pRecord, uint32_t offset,
   uint32_t bytes, uint8_t value)
```

Parameters

→ *`pRecord`*
Pointer to locked data record (chunk pointer).

→ *`offset`*
Offset within record to start writing.

→ *`bytes`*
Number of bytes to write.

→ *`value`*
Byte value to write.

Returns Returns `errNone` if no error. May display a fatal error message if the record pointer is invalid or the function overwrites the record.

Comments Must be used to write to Data Manager records because the data storage area is write-protected.

See Also `DmWrite()`

DmSetDatabaseInfo Function

Purpose Set information about a database.

Declared In `DataMgr.h`

Prototype

```
status_t DmSetDatabaseInfo (DatabaseID dbID,
   DmDatabaseInfoPtr pDatabaseInfo)
```

Parameters

→ *`dbID`*
Database ID of the database.

→ *`pDatabaseInfo`*
Pointer to a structure that contains references to the new database information. See `DmDatabaseInfoType` for a description of the data structure.

Returns Returns `errNone` if no error or one of the following if an error occurred:

`dmErrInvalidDatabaseName`
: The name you've specified for the database is invalid.

`dmErrAlreadyExists`
: Another database with the same name already exists.

`dmErrInvalidParam`
: The function received an invalid parameter.

Comments When this call changes `appInfoID` or `sortInfoID`, the old chunk ID (if any) is marked as an orphaned chunk[1] and the new chunk ID is un-orphaned. Consequently, you shouldn't replace an existing `appInfoID` or `sortInfoID` if that chunk has already been attached to another database.

Call this function to set any or all information about a database except for the database ID. This function sets the new value for any non-`NULL` field in the *`pDatabaseInfo`* structure.

See Also DmDatabaseInfo(), DmOpenDatabaseInfoV50(), DmFindDatabase(), DmGetNextDatabaseByTypeCreator(), TimDateTimeToSeconds()

1. An "orphaned chunk" is one that is allocated in the storage heap, but to which nothing refers. If the orphaned chunk is not put into a database as a record, an Application Info block, or the like, and if the application doesn't keep track of it—in a global variable, perhaps—it could get lost. If the application doesn't get around to freeing the chunk before it quits or crashes, or before the device is reset, that storage will be forever unusable: the user can't delete it since the user only deletes databases.

   During a soft reset, the OS walks through the storage heap and frees any orphaned chunks that it finds. Since most users reset only rarely, however, you shouldn't rely on this happening.

DmSetDatabaseInfoV50 Function

Purpose Set information about a database.

Declared In `DataMgr.h`

Prototype

```
status_t DmSetDatabaseInfoV50 (uint16_t cardNo,
   LocalID dbID, const char *nameP,
   uint16_t *attributesP, uint16_t *versionP,
   uint32_t *crDateP, uint32_t *modDateP,
   uint32_t *bckUpDateP, uint32_t *modNumP,
   LocalID *appInfoIDP, LocalID *sortInfoIDP,
   uint32_t *typeP, uint32_t *creatorP)
```

Parameters

→ *cardNo*
Card number the database resides on.

→ *dbID*
Database ID of the database.

→ *nameP*
Pointer to the new name of the database, or `NULL`. A database name can be up to 32 ASCII bytes long, including the null terminator (as specified by `dmDBNameLength`). Database names must use only 7-bit ASCII characters (0x20 through 0x7E).

→ *attributesP*
Pointer to new attributes variable, or `NULL`. See "Database Attributes" for a list of possible values.

→ *versionP*
Pointer to new version, or `NULL`.

→ *crDateP*
Pointer to new creation date variable, or `NULL`. Specify the value as a number of seconds since Jan. 1, 1904.

→ *modDateP*
Pointer to new modification date variable, or `NULL`. Specify the value as a number of seconds since Jan. 1, 1904.

→ *bckUpDateP*
Pointer to new backup date variable, or `NULL`. Specify the value as a number of seconds since Jan. 1, 1904.

→ *modNumP*
Pointer to new modification number variable, or `NULL`.

→ *`appInfoIDP`*
Pointer to new `appInfoID`, or `NULL`.

→ *`sortInfoIDP`*
Pointer to new `sortInfoID`, or `NULL`.

→ *`typeP`*
Pointer to new `type`, or `NULL`.

→ *`creatorP`*
Pointer to new `creator`, or `NULL`.

Returns Returns `errNone` if no error or one of the following if an error occurred:

`dmErrInvalidDatabaseName`
The name you've specified for the database is invalid.

`dmErrAlreadyExists`
Another database with the same name already exists.

`dmErrInvalidParam`
The function received an invalid parameter.

Comments When this call changes `appInfoID` or `sortInfoID`, the old chunk ID (if any) is marked as an orphaned chunk[2] and the new chunk ID is un-orphaned. Consequently, you shouldn't replace an existing `appInfoID` or `sortInfoID` if that chunk has already been attached to another database.

Call this function to set any or all information about a database except for the card number and database ID. This function sets the new value for any non-`NULL` parameter.

2. An "orphaned chunk" is one that is allocated in the storage heap, but to which nothing refers. If the orphaned chunk is not put into a database as a record, an Application Info block, or the like, and if the application doesn't keep track of it—in a global variable, perhaps—it could get lost. If the application doesn't get around to freeing the chunk before it quits or crashes, or before the device is reset, that storage will be forever unusable: the user can't delete it since the user only deletes databases.

During a soft reset, the OS walks through the storage heap and frees any orphaned chunks that it finds. Since most users reset only rarely, however, you shouldn't rely on this happening.

When setting database attributes, note that the following are system attributes that cannot be set—they are read-only:

`dmHdrAttrResDB`

`dmHdrAttrSchema`

`dmHdrAttrSecure`

`dmHdrAttrOpen`

Compatibility This function is provided for compatibility purposes only. Although it could be used to set information in an extended database, it operates as on previous versions of Palm OS in that the given database name must be unique. Palm OS Cobalt applications—particularly those that are operating on extended databases—will most likely want to use `DmSetDatabaseInfo()` instead.

See Also DmSetDatabaseInfo(), DmDatabaseInfo(), DmOpenDatabaseV50(), DmFindDatabase(), DmGetNextDatabaseByTypeCreator(), TimDateTimeToSeconds()

DmSetDatabaseProtection Function

Purpose Increment or decrement the database's protection count.

Declared In `DataMgr.h`

Prototype
```
status_t DmSetDatabaseProtection
   (DatabaseID dbID, Boolean protect)
```

Parameters

→ *`dbID`*
Database ID of the database.

→ *`protect`*
If `true`, the protection count is incremented. If `false`, the protection count is decremented.

Returns Returns `errNone` if the protection count was updated, or one of the following if an error occurred:

`memErrCardNotPresent`
The specified card can't be found.

`dmErrROMBased`
You've attempted to delete or modify a ROM-based database.

`dmErrCantFind`
: The specified database can't be found.

`memErrNotEnoughSpace`
: A memory error occurred.

`dmErrDatabaseNotProtected`

Comments This function can be used to prevent a database from being deleted (pass `true` for the *`protect`* parameter). All "true" calls should be balanced by "false" calls before the application terminates.

Use this function to keep a particular record or resource in a database locked down without having to keep the database open. Note that because protection counts are kept in the dynamic heap, all databases are "unprotected" at system reset.

If the database is a resource database that has an overlay associated with it for the current locale, the overlay is also protected or unprotected by this function.

DmSetFallbackOverlayLocale Function

Purpose Set the fallback overlay locale: the locale used when the Data Manager attempts to open an overlay locale for which no valid overlay exists.

Declared In `DataMgr.h`

Prototype

```
status_t DmSetFallbackOverlayLocale
    (const LmLocaleType *fallbackLocale)
```

Parameters → *`fallbackLocale`*
: Pointer to a structure identifying the fallback overlay locale.

Returns Returns `errNone` if the fallback overlay locale was successfully set, or one of the following if an error occurred:

`dmErrInvalidParam`
: The function received an invalid parameter.

`dmErrUnknownLocale`
: The specified locale is unknown to the operating system.

Comments The fallback overlay locale is used by the Data Manager when it attempts to automatically open an overlay using the overlay locale, but no valid overlay exists, and the base probably has been stripped.

See Also `DmGetFallbackOverlayLocale()`, `DmSetOverlayLocale()`

DmSetOverlayLocale Function

Purpose Set the Data Manager’s overlay locale: the locale used by the Data Manager when it attempts to automatically open overlays.

Declared In `DataMgr.h`

Prototype
```
status_t DmSetOverlayLocale
   (const LmLocaleType *overlayLocale)
```

Parameters

→ *`overlayLocale`*
Pointer to an `LmLocaleType` structure containing the overlay locale.

Returns Returns `errNone` if the overlay locale was successfully set, or one of the following if an error occurred:

`dmErrInvalidParam`
The function received an invalid parameter.

`dmErrUnknownLocale`
The specified locale is unknown to the operating system.

See Also `DmGetOverlayLocale()`, `DmSetFallbackOverlayLocale()`

DmSetRecordAttr Function

Purpose Set the attributes of a record.

Declared In `DataMgr.h`

Prototype
```
status_t DmSetRecordAttr (DmOpenRef dbRef,
   uint16_t index, uint8_t *pAttr)
```

Parameters

→ *`dbRef`*
`DmOpenRef` to an open database.

→ *`index`*
Index of the record for which attributes are being set.

→ *`pAttr`*
Pointer to the new attributes for the record. See "Non-Schema Database Record Attributes" on page 108 for a description of the attributes. Note that you can only set those attributes not included in the definition of `dmSysOnlyRecAttrs`.

Returns Returns `errNone` if the attributes were successfully set, or one of the following if an error occurred:

`dmErrReadOnly`
You've attempted to write to or modify a database that is open in read-only mode.

`dmErrNotRecordDB`
You've attempted to perform a record function on a resource database.

`dmErrIndexOutOfRange`
The specified index is out of range.

See Also `DmGetRecordAttr()`

DmSetRecordCategory Function

Purpose Set the category information for a record.

Declared In `DataMgr.h`

Prototype
```
status_t DmSetRecordCategory (DmOpenRef dbRef,
   uint16_t index, uint8_t *pCategory)
```

Parameters
→ *`dbRef`*
`DmOpenRef` to an open database.

→ *`index`*
Index of the record for which the category information is being set.

→ *`pCategory`*
Pointer to the new category information for the record.

Returns Returns `errNone` if the category information was successfully set, or one of the following if an error occurred:

`dmErrReadOnly`
    You've attempted to write to or modify a database that is open in read-only mode.

`dmErrNotRecordDB`
    You've attempted to perform a record function on a resource database.

`dmErrIndexOutOfRange`
    The specified index is out of range.

See Also `DmGetRecordCategory()`

DmSetRecordID Function

Purpose Set the unique ID of a record.

Declared In `DataMgr.h`

Prototype
```
status_t DmSetRecordID (DmOpenRef dbRef,
   uint16_t index, uint32_t *pUID)
```

Parameters

→ *`dbRef`*
    `DmOpenRef` to an open database.

→ *`index`*
    Record index for which to set the unique ID.

→ *`pUID`*
    Pointer to the new unique ID.

Returns Returns `errNone` if the record ID was set successfully, or one of the following if an error occurred:

`dmErrInvalidParam`
    The function received an invalid parameter.

`dmErrNotRecordDB`
    You've attempted to perform a record function on a resource database.

`dmErrIndexOutOfRange`
    The specified index is out of range.

`dmErrInvalidID`
    The supplied record ID is already in use.

Comments The Data Manager guarantees that a record ID's uniqueness is maintained after such a call. If the supplied record ID is already in use by another record, this function returns `dmErrInvalidID`.

See Also `DmGetRecordID()`, `DmSetRecordInfoV50()`

DmSetRecordInfoV50 Function

Purpose Set record information stored in the database header.

Declared In `DataMgr.h`

Prototype

```
status_t DmSetRecordInfoV50 (DmOpenRef dbRef,
   uint16_t index, uint16_t *pAttr,
   uint32_t *pUID)
```

Parameters

→ *dbRef*
: `DmOpenRef` to an open database.

→ *index*
: Index of record.

→ *pAttr*
: Pointer to new attribute variable, or `NULL` if you don't want to change any of the record's attributes. See "Non-Schema Database Record Attributes" for a list of possible values.

→ *pUID*
: Pointer to new unique ID, or `NULL` if you don't want to change the record's unique ID.

Returns Returns `errNone` if no error, or one of the following if an error occurred:

`dmErrReadOnly`
: You've attempted to write to or modify a database that is open in read-only mode.

`dmErrNotRecordDB`
: You've attempted to perform a record function on a resource database.

`dmErrIndexOutOfRange`
: The specified index is out of range.

Some releases may display a fatal error message instead of returning the error code.

Comments Sets information about a record. This function cannot be used to set the `dmRecAttrBusy` bit; instead, use `DmGetRecord()` to set the bit and `DmReleaseRecord()` to clear it.

Normally, the unique ID for a record is automatically created by the Data Manager when a record is created using `DmNewRecord()`, so an application would not typically change the unique ID.

Compatibility Provided for compatibility purposes only. Palm OS Cobalt applications should use `DmSetRecordAttr()` and/or `DmSetRecordID()` instead.

See Also DmSetRecordAttr(), DmSetRecordID(), DmGetRecordAttr(), DmGetRecordID(), DmRecordInfoV50()

DmSetResourceInfo Function

Purpose Set information on a given resource.

Declared In `DataMgr.h`

Prototype
```
status_t DmSetResourceInfo (DmOpenRef dbRef,
   uint16_t index, DmResourceType *pResType,
   DmResourceID *pResID)
```

Parameters

→ *`dbRef`*
`DmOpenRef` to an open database.

→ *`index`*
Index of resource to set info for.

→ *`pResType`*
Pointer to new `resType` (resource type), or `NULL`.

→ *`pResID`*
Pointer to new resource ID, or `NULL`.

Returns Returns `errNone` if no error, or one of the following if an error occurred:

`dmErrIndexOutOfRange`
The specified index is out of range.

`dmErrReadOnly`
You've attempted to write to or modify a database that is open in read-only mode.

May display a fatal error message if the database is not a resource database.

Comments Use this function to set all or a portion of the information on a particular resource. Any or all of the new info pointers can be `NULL`. If not `NULL`, the type and ID of the resource are changed to `*`*`pResType`* and `*`*`pResID`*.

DmStrCopy Function

Purpose Copies a string to a record within a database that is open for writing.

Declared In `DataMgr.h`

Prototype

```
status_t DmStrCopy (void *pRecord,
   uint32_t offset, const void *pSrc)
```

Parameters

↔ *`pRecord`*
Pointer to data record (chunk pointer).

→ *`offset`*
Offset within record to start writing.

→ *`pSrc`*
Pointer to null-terminated string.

Returns Returns `errNone` if no error. May display a fatal error message if the record pointer is invalid or the function overwrites the record.

Comments This is one of the functions that must be used to write to Data Manager records; because the data storage area is write-protected, you cannot write to it directly. This function checks the validity of the chunk pointer for the record to ensure that writing the record will not exceed the chunk bounds. `DmStrCopy()` is a convenience method that determines the size of the supplied string and then simply calls `DmWrite()`.

See Also DmSet()

DmWrite Function

Purpose Copies a specified number of bytes to a record within a database that is open for writing.

Declared In `DataMgr.h`

Prototype
```
status_t DmWrite (void *pRecord, uint32_t offset,
   const void *pSrc, uint32_t bytes)
```

Parameters

↔ *`pRecord`*
Pointer to locked data record (chunk pointer).

→ *`offset`*
Offset within record to start writing.

→ *`pSrc`*
Pointer to data to copy into record.

→ *`bytes`*
Number of bytes to write.

Returns Returns `errNone` if no error. May display a fatal error message if the record pointer is invalid or the function overwrites the record.

Comments This is one of the functions that must be used to write to Data Manager records; because the data storage area is write-protected, you cannot write to it directly. This function checks the validity of the chunk pointer for the record to ensure that writing the record will not exceed the chunk bounds.

See Also `DmStrCopy()`, `DmSet()`

DmWriteCheckV50 Function

Purpose Check the parameters of a write operation to a classic database data storage chunk before actually performing the write.

Declared In `DataMgr.h`

Prototype
```
status_t DmWriteCheckV50 (void *pRecord,
   uint32_t offset, uint32_t bytes)
```

Parameters

→ *`pRecord`*
Locked pointer to the record handle.

→ *`offset`*
Offset into record to start writing.

→ *bytes*
Number of bytes to write.

Returns Returns `errNone` if no error; returns `dmErrNotValidRecord` or `dmErrWriteOutOfBounds` if an error occurred.

Compatibility This function operates only with classic databases, and is provided only for compatibility purposes. Palm OS Cobalt applications should go ahead and write the data using a function such as `DmWrite()`, checking the returned status code to determine if an error occurred.

Application-Defined Functions

DmCompareFunctionType Function

Purpose Compares two records in a classic database.

Declared In `DataMgr.h`

Prototype
```
int16_t DmCompareFunctionType (void *rec1P,
   void *rec2P, int16_t other,
   DmSortRecordInfoPtr rec1SortInfoP,
   DmSortRecordInfoPtr rec2SortInfoP,
   MemHandle appInfoH)
```

Parameters

→ *rec1P*
Pointer to the first record to compare.

→ *rec2P*
Pointer to the second record to compare.

→ *other*
Any other custom information you want passed to the comparison function. This parameter is often used to indicate a sort direction (ascending or descending).

→ *rec1SortInfoP*
Pointer to a `DmSortRecordInfoType` structure that specifies unique sorting information for the first record.

→ *rec2SortInfoP*
Pointer to a `DmSortRecordInfoType` structure that specifies unique sorting information for the second record.

→ *`appInfoH`*
A handle to the database's Application Info block.

Returns Your implementation of this function should return:

- 0 if `rec1` = `rec2`.
- < 0 if `rec1` < `rec2`.
- > 0 if `rec1` > `rec2`.

Comments This function is used to sort the records in a database. It is specifically called by `DmGetRecordSortPosition()`, `DmInsertionSort()`, and `DmQuickSort()`.

5

File Stream

This chapter provides reference material for the File Stream API. It is organized as follows:

The header file `FileStream.h` declares the API that this chapter describes.

For more information on file streams in Palm OS®, see Chapter 2, "Palm OS Databases," on page 11.

File Stream Structures and Types

FileHand Typedef

Purpose Handle to an open file stream.

Declared In `FileStream.h`

Prototype `typedef MemHandle FileHand`

Comments Open a file stream and receive a handle to it with `FileOpen()`.

File Stream Constants

File Stream Error Codes

Purpose Error codes returned by the various File Stream functions.

Declared In `FileStream.h`

Constants

`#define fileErrCloseError (fileErrorClass | 12)`
Error closing the stream.

`#define fileErrCorruptFile (fileErrorClass | 3)`
The stream is corrupted, invalid, or not a stream.

`#define fileErrCreateError (fileErrorClass | 7)`
Couldn't create new stream.

`#define fileErrEOF (fileErrorClass | 16)`
End-of-File error.

`#define fileErrInUse (fileErrorClass | 9)`
Stream couldn't be opened or deleted because it is in use.

`#define fileErrInvalidDescriptor (fileErrorClass | 11)`
Invalid file descriptor (`FileHandle`).

`#define fileErrInvalidParam (fileErrorClass | 2)`
Invalid parameter value passed.

`#define fileErrIOError (fileErrorClass | 15)`
Generic I/O error.

`#define fileErrMemError (fileErrorClass | 1)`
Out of memory error.

`#define fileErrNotFound (fileErrorClass | 4)`
Couldn't find the stream.

`#define fileErrNotStream (fileErrorClass | 17)`
Attempted to open an entity that is not a stream.

`#define fileErrOpenError (fileErrorClass | 8)`
Generic open error.

`#define fileErrOutOfBounds (fileErrorClass | 13)`
Attempted operation went out of bounds of the stream.

`#define fileErrPermissionDenied (fileErrorClass | 14)`
: Couldn't write to a stream open for read-only access.

`#define fileErrReadOnly (fileErrorClass | 10)`
: Couldn't open in write mode because existing stream is read-only.

`#define fileErrReplaceError (fileErrorClass | 6)`
: Couldn't replace existing stream.

`#define fileErrTypeCreatorMismatch (fileErrorClass | 5)`
: Type and/or creator not what was specified.

Primary Open Modes

Purpose Specify the mode in which a file stream is opened.

Declared In `FileStream.h`

Constants

```
#define fileModeAllFlags ( fileModeReadOnly |
  fileModeReadWrite | fileModeUpdate |
  fileModeAppend | fileModeLeaveOpen |
  fileModeExclusive | fileModeAnyTypeCreator |
  fileModeTemporary | fileModeDontOverwrite )
```
The complete set of file stream open modes.

`#define fileModeAppend (0x10000000UL)`
: Open/create for read/write, always writing to the end of the stream

`#define fileModeReadOnly (0x80000000UL)`
: Open for read-only access

`#define fileModeReadWrite (0x40000000UL)`
: Open/create for read/write access, discarding any previous version of stream

`#define fileModeUpdate (0x20000000UL)`
: Open/create for read/write, preserving previous version of stream if it exists

Comments For each file stream, you must pass to the `FileOpen()` function only one of the primary mode selectors listed. Note that you can combine the primary mode selector with one or more secondary mode selectors for additional control.

Secondary Open Modes

Purpose Additional mode selectors that can be OR'd with a primary mode selector to provide additional control.

Declared In `FileStream.h`

Constants `#define fileModeAnyTypeCreator (0x02000000UL)`
Accept any type/creator when opening or replacing an existing stream. Normally, the `FileOpen()` function opens only streams having the specified creator and type. Setting this option enables the `FileOpen()` function to open streams having a type or creator other than those specified.

`#define fileModeDontOverwrite (0x00800000UL)`
Prevents `fileModeReadWrite` from discarding an existing stream having the same name; may only be specified together with `fileModeReadWrite`.

`#define fileModeExclusive (0x04000000UL)`
No other application can open the stream until the application that opened it in this mode closes it.

`#define fileModeLeaveOpen (0x08000000UL)`
Leave stream open when application quits. Palm OS Cobalt applications should not use this option.

`#define fileModeTemporary (0x01000000UL)`
Delete the stream automatically when it is closed. For more information, see Comment section of `FileOpen()` function description.

Miscellaneous File Stream Constants

Purpose The File Stream APIs also include the following `#defines`.

Declared In `FileStream.h`

Constants `#define fileNullHandle ((FileHand)0)`
An invalid file handle.

FileOpEnum Enum

Purpose Control operations that can be performed on a file stream with `FileControl()`.

Declared In `FileStream.h`

Constants `fileOpNone = 0`

No-op.

`fileOpDestructiveReadMode`

Enter destructive read mode, and rewind stream to its beginning. Once in this mode, there is no turning back: stream's contents after closing (or crash) are undefined.

Destructive read mode deletes blocks as data are read, thus freeing storage automatically. Once in destructive read mode, you cannot re-use the file stream—the contents of the stream are undefined after it is closed or after a crash.

Writing to files opened without write access or those that are in destructive read state is not allowed; thus, you cannot call the `FileWrite()`, `FileSeek()`, or `FileTruncate()` functions on a stream that is in destructive read mode. One exception to this rule applies to streams that were opened in "write + append" mode and then switched into destructive read state. In this case, the `FileWrite()` function can append data to the stream, but it also preserves the current stream position so that subsequent reads pick up where they left off (you can think of this as a pseudo-pipe).

`ARGUMENTS:`

`stream` = open stream handle

`valueP` = NULL

`valueLenP` = NULL

`RETURNS:`

zero on success;

`fileErr`... on error

`fileOpGetEOFStatus`

Get end-of-file status (like C runtime's `feof`) (err = `fileErrEOF`). Indicates end of file condition. Use `FileClearerr()` to clear this error status.

`ARGUMENTS:`

`stream` = open stream handle

`valueP` = NULL

`valueLenP` = NULL

`RETURNS:`

zero if not end of file;

non-zero if end of file

`fileOpGetLastError`

Get error code from last operation on stream, and clear the last error code value. Doesn’t change status of EOF or I/O errors —use `FileClearerr()` to reset all error codes.

`ARGUMENTS:`

`stream` = open stream handle

`valueP` = NULL

`valueLenP` = NULL

`RETURNS:`

Error code from last file stream operation

`fileOpClearError`

Clear I/O and EOF error status and last error.

`ARGUMENTS:`

`stream` = open stream handle

`valueP` = NULL

`valueLenP` = NULL

`RETURNS:`

zero on success; `fileErr`... on error

`fileOpGetIOErrorStatus`

Get I/O error status (like C runtime's `ferror`). Use `FileClearerr()` to clear this error status.

`ARGUMENTS:`

`stream` = open stream handle

`valueP` = NULL

`valueLenP` = NULL

`RETURNS:`

zero if not I/O error;

non-zero if I/O error is pending.

`fileOpGetCreatedStatus`

Find out whether file was created by `FileOpen()` function

`ARGUMENTS:`

`stream` = open stream handle

`valueP` = Pointer to Boolean

`valueLenP` = Pointer to `Int32` variable set to sizeof(Boolean)

`RETURNS:`

zero on success; `fileErr`... on error. The Boolean variable will be set to non-zero if the file was created.

`fileOpGetOpenDbRef`

Get the open database reference (handle) of the underlying database that implements the stream (`NULL` if none); this is needed for performing Palm OS-specific operations on the underlying database, such as changing or getting creator and type, version, backup/reset bits, and so on.

`ARGUMENTS:`

`stream` = open stream handle

`valueP` = Pointer to `DmOpenRef` variable

`valueLenP` = Pointer to Int32 variable set to sizeof(DmOpenRef)

`RETURNS:`

zero on success; `fileErr`... on error. The `DmOpenRef` variable will be set to the file's open db reference that may be passed to Data Manager calls;

WARNING! Do not make any changes to the data of the underlying database—doing so will corrupt the file stream.

`fileOpFlush`

Flush any cached data to storage.

`ARGUMENTS:`

`stream` = open stream handle

`valueP` = NULL

`valueLenP` = NULL

`RETURNS:`

zero on success; `fileErr`... on error;

`fileOpLAST`
Not an actual operator, this value simply identifies the end of the list of file control operations.

FileOriginEnum Enum

Purpose File positions to which an offset is added (or subtracted, if the offset is negative) to get a seek position within the file.

Declared In `FileStream.h`

Constants `fileOriginBeginning = 1`
From the beginning (first data byte of file).

`fileOriginCurrent`
From the current position.

`fileOriginEnd`
From the end of file (one position beyond last data byte).

Comments Supply one of these values to `FileSeek()`.

File Stream Functions and Macros

FileClearerr Macro

Purpose Clear I/O error status, end of file error status, and last error.

Declared In `FileStream.h`

Prototype `#define FileClearerr (__stream__)`

Parameters → `__stream__`
Handle to an open stream.

Returns Returns `errNone` if no error, or a `fileErr` code if an error occurs. See the section "File Stream Error Codes" for more information.

See Also `FileGetLastError()`, `FileRewind()`

FileClose Function

Purpose Close the file stream and destroy its handle. If the stream was opened with `fileModeTemporary`, it is deleted upon closing.

Declared In `FileStream.h`

Prototype

```
status_t FileClose (FileHand stream)
```

Parameters

→ *`stream`*
Handle to an open stream.

Returns Returns `errNone` if no error, or a `fileErr` code if an error occurs. See the section "File Stream Error Codes" for more information.

FileControl Function

Purpose Perform a specified operation on a file stream.

Declared In `FileStream.h`

Prototype

```
status_t FileControl (FileOpEnum op,
    FileHand stream, void *valueP,
    int32_t *valueLenP)
```

Parameters

→ *`op`*
The operation to perform, and its associated formal parameters. See "FileOpEnum" on page 243 for a list of possible values.

→ *`stream`*
Open stream handle if required for file stream operation.

↔ *`valueP`*
Pointer to value or buffer, as required. This parameter is defined by the selector passed as the value of the *op* parameter. For details, see "FileOpEnum" on page 243.

↔ *`valueLenP`*
Pointer to value or buffer, as required. This parameter is defined by the selector passed as the value of the *`op`* parameter. For details, see "FileOpEnum" on page 243.

Returns Returns either a value defined by the selector passed as the argument to the *`op`* parameter, or an error code resulting from the requested operation.

Comments Normally, you do not call the `FileControl()` function yourself; it is called for you by most of the other file streaming functions and macros to perform common file streaming operations. You can call `FileControl()` yourself to enable specialized read modes.

See Also `FileClearerr()`, `FileEOF()`, `FileError()`, `FileFlush()`, `FileGetLastError()`, `FileRewind()`

FileDelete Function

Purpose Deletes the specified file stream from the specified card. Only a closed stream may be passed to this function.

Declared In `FileStream.h`

Prototype

```
status_t FileDelete (const char *nameP,
   uint32_t creator)
```

Parameters

→ *`nameP`*
Name of the stream to delete.

→ *`creator`*
Creator of the file stream to delete.

Returns Returns `errNone` if no error, or a `fileErr` code if an error occurs. See the section "File Stream Error Codes" for more information.

See Also `FileOpen()`

FileDeleteV50 Function

Purpose Deletes the specified file stream from the specified card. Only a closed stream may be passed to this function.

Declared In `FileStream.h`

Prototype
```
status_t FileDeleteV50 (uint16_t cardNo,
   const char *nameP)
```

Parameters

→ *`cardNo`*
Card on which the file stream to delete resides.

→ *`nameP`*
Name of the stream to delete.

Returns Returns `errNone` if no error, or a `fileErr` code if an error occurs. See the section "File Stream Error Codes" for more information.

Compatibility This function is only provided for compatibility with previous versions of Palm OS; the *`cardNo`* parameter is ignored.

See Also `FileOpen()`

FileDmRead Macro

Purpose Reads data from a file stream into a chunk, record, or resource residing in a database.

Declared In `FileStream.h`

Prototype
```
#define FileDmRead (stream, startOfDmChunkP,
   destOffset, objSize, numObj, errP)
```

Parameters

→ *`stream`*
Handle to an open stream.

→ *`startOfDmChunkP`*
Pointer to beginning of chunk, record or resource residing in a database.

→ *`destOffset`*
Offset from `startOfDmChunkP` (base pointer) to the destination area (must be >= 0).

→ *`objSize`*
Size of each stream object to read.

→ *`numObj`*
Number of stream objects to read.

← *`errP`*
Pointer to a variable that is to hold the error code returned by this function. Pass `NULL` to ignore. See the section "File Stream Error Codes" for more information.

Returns The number of whole objects that were read. Note that the number of objects actually read may be less than the number requested.

Comments When the number of objects actually read is less than the number requested, you may be able to determine the cause of this result by examining the return value of the *`errP`* parameter or by calling the `FileGetLastError()` function. If the cause is insufficient data in the stream to satisfy the full request, the current stream position is at end-of-file and the "end of file" indicator is set. If a non-`NULL` pointer was passed as the value of the *`errP`* parameter when `FileDmRead` was used and an error was encountered, `*`*`errP`* holds a non-zero error code when the function returns. In addition, the `FileError()` and `FileEOF()` functions may be used to check for I/O errors.

See Also FileRead(), FileError(), FileEOF()

FileEOF Macro

Purpose Get end-of-file status (err = `fileErrEOF` indicates end of file condition).

Declared In `FileStream.h`

Prototype `#define FileEOF (`*`__stream__`*`)`

Parameters → *`__stream__`*
Handle to an open stream.

Returns Returns 0 if not at the end of file, `fileErrEOF` if at the end of file, or an error code otherwise. See the section "File Stream Error Codes" for more information.

Comments This macro's behavior is similar to that of the `feof` function provided by the C programming language runtime library.

Use `FileClearerr()` to clear the I/O error status.

See Also `FileClearerr()`, `FileGetLastError()`, `FileRewind()`

FileError Macro

Purpose Get I/O error status.

Declared In `FileStream.h`

Prototype `#define FileError (__`*`stream`*`__)`

Parameters → `__`*`stream`*`__`
Handle to an open stream.

Returns Returns `errNone` if no error, and non-zero if an I/O error indicator has been set for this stream. See the section "File Stream Error Codes" for more information.

Comments This macro's behavior is similar to that of the C programming language's `ferror` runtime function.

Use `FileClearerr()` to clear the I/O error status.

See Also `FileClearerr()`, `FileGetLastError()`, `FileRewind()`

FileFlush Macro

Purpose Flush cached data to storage.

Declared In `FileStream.h`

Prototype `#define FileFlush (__`*`stream`*`__)`

Parameters → `__`*`stream`*`__`
Handle to an open stream.

Returns Returns `errNone` if no error, or a `fileErr` code if an error occurs. See the section "File Stream Error Codes" for more information.

Comments It is not always necessary to call this macro explicitly—certain operations flush the contents of a stream automatically; for example, streams are flushed when they are closed. Because this macro's behavior is similar to that of the `fflush()` function provided by the C programming language runtime library, you only need to call it explicitly under circumstances similar to those in which you would call `fflush` explicitly.

FileGetLastError Macro

Purpose Get error code from last operation on file stream, and clear the last error code value (will not change end of file or I/O error status -- use `FileClearerr()` to reset all error codes)

Declared In `FileStream.h`

Prototype `#define FileGetLastError (__stream__)`

Parameters → *`__stream__`*
Handle to an open stream.

Returns Returns the error code returned by the last file stream operation. See the section "File Stream Error Codes" for more information.

See Also `FileClearerr()`, `FileEOF()`, `FileError()`

FileOpen Function

Purpose Open existing file stream or create an open file stream (an extended database) for I/O in the specified mode.

Declared In `FileStream.h`

Prototype
```
FileHand FileOpen (const char *nameP,
   uint32_t type, uint32_t creator,
   uint32_t openMode, status_t *errP)
```

Parameters → *`nameP`*
Pointer to the name of the extended database to open or create as a file stream. This value must be a valid name—no wildcards allowed, and composed only of 7-bit ASCII characters—and must not be `NULL`.

→ *`type`*
File type of stream to open or create. Pass `0` for wildcard, in which case `sysFileTFileStream` is used if the stream needs to be created and `fileModeTemporary` is not specified. If *`type`* is `0` and `fileModeTemporary` is specified, then `sysFileTTemp` is used for the file type of the stream this function creates.

→ *`creator`*
Creator of stream to open or create. Pass `0` for wildcard, in which case the current application's creator ID is used for the creator of the stream this function creates.

→ *`openMode`*
Mode in which to open the file stream. You must specify only one primary mode selector. Additionally, you can use the bitwise inclusive OR operator to append one or more secondary mode selectors to the primary mode selector. See "Primary Open Modes" and "Secondary Open Modes" for the list of possible values.

← *`errP`*
Pointer to a variable that is to hold the error code returned by this function. Pass `NULL` to ignore. See the section "File Stream Error Codes" for a list of error codes.

Returns If successful, returns a handle to an open file stream; otherwise, returns `0`.

In some cases, `FileOpen()` returns a non-zero value when it has failed to open a file; thus, it is always a good idea to check the *`errP`* parameter value to determine if an error has occurred.

Comments **IMPORTANT:** Previous versions of Palm OS didn't enforce the requirement that database names passed to `FileOpen()` be composed only of 7-bit ASCII characters. Palm OS Cobalt requires that this be so.

The `fileModeReadOnly`, `fileModeReadWrite`, `fileModeUpdate`, and `fileModeAppend` modes are mutually exclusive—pass only one of them to the `FileOpen()` function!

When the `fileModeTemporary` open mode is used and the file type passed to `FileOpen()` is `0`, the `FileOpen()` function uses `sysFileTTemp` (defined in `SystemMgr.rh)` for the file type, as recommended. In future versions of Palm OS, this configuration will enable the automatic cleanup of undeleted temporary files after a system crash. Automatic post-crash cleanup is not implemented in current versions of Palm OS.

To open a file stream even if it has a different type and creator than specified, pass the `fileModeAnyTypeCreator` selector as a flag in the `openMode` parameter to the `FileOpen()` function.

The `fileModeLeaveOpen` mode is an esoteric option that most applications should not use. It may be useful for a library that needs to open a stream from the current application's context and keep it open even after the current application quits. By default, Palm OS automatically closes all databases that were opened in a particular application's context when that application quits. The `fileModeLeaveOpen` option overrides this default behavior.

FileOpenV50 Function

Purpose Open existing file stream or create an open file stream (a classic database) for I/O in the mode specified by the `openMode` parameter.

Declared In `FileStream.h`

Prototype
```
FileHand FileOpenV50 (uint16_t cardNo,
   const char *nameP, uint32_t type,
   uint32_t creator, uint32_t openMode,
   status_t *errP)
```

Parameters

→ *`cardNo`*
Card on which the file stream to open resides.

→ *`nameP`*
Pointer to the name of the classic database to open or create as a file stream. This value must be a valid name—no wildcards allowed, and composed only of 7-bit ASCII characters—and must not be `NULL`.

→ *`type`*
File type of stream to open or create. Pass `0` for wildcard, in which case `sysFileTFileStream` is used if the stream needs to be created and `fileModeTemporary` is not specified. If *`type`* is `0` and `fileModeTemporary` is specified, then `sysFileTTemp` is used for the file type of the stream this function creates.

→ *`creator`*
Creator of stream to open or create. Pass `0` for wildcard, in which case the current application's creator ID is used for the creator of the stream this function creates.

→ *openMode*

Mode in which to open the file stream. You must specify only one primary mode selector. Additionally, you can use the bitwise inclusive OR operator to append one or more secondary mode selectors to the primary mode selector. See "Primary Open Modes" and "Secondary Open Modes" for the list of possible values.

← *errP*

Pointer to a variable that is to hold the error code returned by this function. Pass `NULL` to ignore. See the section "File Stream Error Codes" for a list of error codes.

Returns If successful, returns a handle to an open file stream; otherwise, returns `0`.

In some cases, on some platforms, `FileOpen()` returns a non-zero value when it has failed to open a file; thus, it is always a good idea to check the *errP* parameter value to determine if an error has occurred.

Comments **IMPORTANT:** Previous versions of Palm OS didn't enforce the requirement that database names passed to `FileOpen()` be composed only of 7-bit ASCII characters. Palm OS Cobalt requires that this be so.

The `fileModeReadOnly`, `fileModeReadWrite`, `fileModeUpdate`, and `fileModeAppend` modes are mutually exclusive—pass only one of them to the `FileOpen()` function!

When the `fileModeTemporary` open mode is used and the file type passed to `FileOpen()` is `0`, the `FileOpen()` function uses `sysFileTTemp` (defined in `SystemMgr.rh)` for the file type, as recommended. In future versions of Palm OS, this configuration will enable the automatic cleanup of undeleted temporary files after a system crash. Automatic post-crash cleanup is not implemented in current versions of Palm OS.

To open a file stream even if it has a different type and creator than specified, pass the `fileModeAnyTypeCreator` selector as a flag in the `openMode` parameter to the `FileOpen()` function.

The `fileModeLeaveOpen` mode is an esoteric option that most applications should not use. It may be useful for a library that needs to open a stream from the current application's context and keep it open even after the current application quits. By default, Palm OS automatically closes all databases that were opened in a particular application's context when that application quits. The `fileModeLeaveOpen` option overrides this default behavior.

Compatibility This function is only provided for compatibility with previous versions of Palm OS; the *`cardNo`* parameter is ignored.

FileRead Macro

Purpose Reads data from a stream into a buffer.

Declared In `FileStream.h`

Prototype
```
#define FileRead (stream, bufP, objSize, numObj,
   errP)
```

Parameters

→ *`stream`*
Handle to an open stream.

→ *`bufP`*
Pointer to a buffer into which data is read

→ *`objSize`*
Size of each stream object to read.

→ *`numObj`*
Number of stream objects to read.

← *`errP`*
Pointer to a variable that is to hold the error code returned by this macro. Pass `NULL` to ignore. See the section "File Stream Error Codes" for a list of error codes.

Returns Returns the number of whole objects that were read. Note that the number of objects actually read may be less than the number requested.

Comments Do not use this macro to read data into a chunk, record or resource residing in a database—you must use the `FileDmRead()` macro for such operations.

When the number of objects actually read is fewer than the number requested, you may be able to determine the cause of this result by examining the return value of the *`errP`* parameter or by calling the `FileGetLastError()` function. If the cause is insufficient data in the stream to satisfy the full request, the current stream position is at end-of-file and the "end of file" indicator is set. If a non-`NULL` pointer was passed as the value of the *`errP`* parameter when the `FileRead()` function was called and an error was encountered, `*`*`errP`* holds a non-zero error code when the function returns. In addition, the `FileError()` and `FileEOF()` functions may be used to check for I/O errors.

See Also `FileDmRead()`

FileReadLow Function

Purpose Reads data from a file into a buffer or a data storage heap-based chunk (record or resource). Use the `FileRead()` and `FileDmRead()` macros instead of calling this function directly.

Declared In `FileStream.h`

Prototype

```
int32_t FileReadLow (FileHand stream,
   void *baseP, int32_t offset,
   Boolean dataStoreBased, int32_t objSize,
   int32_t numObj, status_t *errP)
```

Parameters

→ *`stream`*
Handle to an open stream.

→ *`baseP`*
Pointer to a buffer into which data is read

→ *`offset`*
Offset into the *`baseP`* buffer marking the place at which the read data is stored.

→ *`dataStoreBased`*
`true` if the buffer is data-store based (that is, if it is a chunk, record or resource residing in a database) or `false` if it is located in the dynamic heap.

→ *`objSize`*
Size of each stream object to read.

→ *`numObj`*
Number of stream objects to read.

← *errP*
Pointer to a variable that is to hold the error code returned by this function. Pass `NULL` to ignore. See the section "File Stream Error Codes" for a list of error codes.

Returns Returns the number of whole objects that were read. Note that the number of objects actually read may be less than the number requested.

Comments Use the `FileRead()` and `FileDmRead()` macros instead of calling this function directly.

FileRewind Macro

Purpose Reset position marker to beginning of stream and clear all error codes.

Declared In `FileStream.h`

Prototype `#define FileRewind (__stream__)`

Parameters → *`__stream__`*
Handle to an open stream.

Returns Returns `errNone` if no error, or a `fileErr` code if an error occurs. See the section "File Stream Error Codes" for more information.

See Also `FileSeek()`, `FileTell()`, `FileClearerr()`, `FileEOF()`, `FileError()`, `FileGetLastError()`

FileSeek Function

Purpose Set current position within a file stream, extending the stream as necessary if it was opened with write access.

Declared In `FileStream.h`

Prototype
```
status_t FileSeek (FileHand stream,
   int32_t offset, FileOriginEnum origin)
```

Parameters → *stream*
Handle to an open stream.

→ *offset*
Position to set, expressed as the number of bytes from *origin*. This value may be positive, negative, or `0`.

→ *`origin`*
: Origin of the position change. Supply one of the values documented under "FileOriginEnum" on page 246.

Returns Returns `errNone` if no error, or a `fileErr` code if an error occurs. See the section "File Stream Error Codes" for more information.

Comments Attempting to seek beyond end-of-file in a read-only stream results in an I/O error.

This function's behavior is similar to that of the `fseek` function provided by the C programming language runtime library.

See Also `FileRewind()`, `FileTell()`

FileTell Function

Purpose Retrieves the current position and, optionally, the file size of a stream.

Declared In `FileStream.h`

Prototype

```
int32_t FileTell (FileHand stream,
   int32_t *fileSizeP, status_t *errP)
```

Parameters

→ *`stream`*
: Handle to an open stream.

← *`fileSizeP`*
: Pointer to variable that receives the size of the stream in bytes. Pass `NULL` to ignore.

← *`errP`*
: Pointer to a variable that is to hold the error code returned by this function. Pass `NULL` to ignore. See the section "File Stream Error Codes" for a list of error codes.

Returns If successful, returns the current position, expressed as an offset in bytes from the beginning of the stream. If an error was encountered, returns `-1`.

Comments The `FileTell()` function can return the size of the input stream; as such, it provides some of the functionality of the standard C library `stat` function. Note, however, that unlike the `stat` function, `FileTell()` requires that the file be open.

See Also `FileRewind()`, `FileSeek()`

FileTruncate Function

Purpose Truncate the file stream to a specified size.

Declared In `FileStream.h`

Prototype
```
status_t FileTruncate (FileHand stream,
   int32_t newSize)
```

Parameters

→ *`stream`*
Handle to an open stream.

→ *`newSize`*
New size; must not exceed current stream size.

Returns Returns `errNone` if no error, or a `fileErr` code if an error occurs. See the section "File Stream Error Codes" for more information.

Comments This function cannot be used on streams that are open in destructive read mode or read-only mode.

See Also `FileTell()`

FileWrite Function

Purpose Write data to a stream.

Declared In `FileStream.h`

Prototype
```
int32_t FileWrite (FileHand stream,
   const void *dataP, int32_t objSize,
   int32_t numObj, status_t *errP)
```

Parameters

→ *`stream`*
Handle to an open stream.

→ *`dataP`*
Pointer to a buffer holding the data to be written.

→ *objSize*
Size of each stream object to write. Must be greater than or equal to `0`.

→ *numObj*
Number of stream objects to write.

← *errP*
Pointer to a variable that is to hold the error code returned by this function. Pass `NULL` to ignore. See the section "File Stream Error Codes" for a list of error codes.

Returns Returns the number of whole objects that were written. Note that the number of objects actually written may be less than the number requested. Should available storage be insufficient to satisfy the entire request, as much of the requested data as possible is written to the stream, which may result in the last object in the stream being incomplete.

Comments Writing to files opened without write access or those that are in destructive read state is not allowed; thus, you cannot call the `FileWrite()`, `FileSeek()`, or `FileTruncate()` functions on a stream that is in destructive read mode. One exception to this rule applies to streams that were opened in "write + append" mode and then switched into destructive read state. In this case, the `FileWrite` function can append data to the stream, but it also preserves the current stream position so that subsequent reads pick up where they left off (you can think of this as a pseudo-pipe).

6

Memory Manager

This chapter describes the Memory Manager APIs. You use these APIs to manipulate memory chunks and memory heaps within Palm OS®.

Note that many of the APIs provided by the Memory Manager exist to simplify the process of porting an application from an earlier version of Palm OS. Palm OS Cobalt applications can make use of the standard C memory management functions—functions such as `malloc()`, `realloc()`, and `free()`—instead.

This chapter is organized as follows:

The header file `MemoryMgr.h` declares the API that this chapter describes.

For more information on the Memory Manager, see Chapter 1, "Memory," on page 3.

Memory Manager Structures and Types

LocalID Typedef

Purpose Chunk identifier.

Declared In `MemoryMgr.h`

Prototype
```
typedef uint32_t LocalID
```

MemHeapInfoType Struct

Purpose Contains information about a dynamic heap.

Declared In `MemoryMgr.h`

Prototype
```
typedef struct MemHeapInfoType {
   uint32_t maxBlockSize;
   uint32_t defaultAlignment;
   void *basePtr;
   uint32_t maxSize;
   uint32_t physMem;
   uint32_t physMemUsed;
   uint32_t physMemUnused;
   uint32_t chunksNum;
   uint32_t memAllocated;
   uint32_t chunksFree;
   uint32_t freeSpace;
   uint32_t freeBytes;
   uint32_t largestBlock;
   uint32_t largestCommitted;
   uint32_t statMaxAllocated;
} MemHeapInfoType
typedef MemHeapInfoType *MemHeapInfoPtr
```

Fields

`maxBlockSize`
The size of the largest chunk that could be potentially allocated.

`defaultAlignment`
The default alignment of memory chunks.

`basePtr`
The base address of the dynamic heap.

`maxSize`
: The amount of virtual address space reserved for the heap.

`physMem`
: The amount of physical memory that could be used to extend the pool of memory chunks.

`physMemUsed`
: The amount of physical memory being used by the dynamic heap.

`physMemUnused`
: The amount of physical memory that could be returned to the operating system.

`chunksNum`
: The number of chunks allocated from the heap.

`memAllocated`
: The amount of memory used by chunks that are not free.

`chunksFree`
: The number of chunks in the dynamic heap that are free.

`freeSpace`
: The amount of uncommitted virtual address space reserved for chunks.

`freeBytes`
: The total number of bytes that could potentially be used to allocate chunks.

`largestBlock`
: The size of the largest memory block that could be allocated from the dynamic heap.

`largestCommitted`
: the size of the largest memory block that could be allocated from the dynamic heap without using additional kernel memory.

`statMaxAllocated`

Comments Use `MemDynHeapGetInfo()` to obtain this information.

Memory Manager Constants

Debug Mode Flags

Purpose These flags indicate or specify the current debug mode for the instance of the Heap Manager local to the calling process.

Declared In `MemoryMgr.h`

Constants

`#define memDebugModeAllHeaps 0x0020`
Obsolete flag. Provided for compatibility purposes only.

`#define memDebugModeCheckOnAll 0x0002`

`#define memDebugModeCheckOnChange 0x0001`

`#define memDebugModeFillFree 0x0010`
When a memory chunk is freed (with either `MemPtrFree()` or `MemHandleFree()`), unused memory will be filled with a default value (currently, `0x55`). Note that only memory that is accessible will be filled: the first 32 bits of free chunk data are reserved for internal use and will never be filled.

`#define memDebugModeNoDMCall 0x0200`
Force the heap library to report all calls that it delegates to the Data Manager. This flag helps you to track down Memory Manager calls that operate on the storage heap—calls that should be changed to reference the corresponding Data Manager functions.

`#define memDebugModeRecordMaxDynHeapUsed memDebugModeRecordMinDynHeapFree`
Records the maximum amount of memory used by the dynamic heap during its lifetime.

`#define memDebugModeRecordMinDynHeapFree 0x0040`
Records the maximum amount of memory used by the dynamic heap during its lifetime.

`#define memDebugModeScrambleOnAll 0x0008`
Obsolete flag. Provided for compatibility purposes only.

`#define memDebugModeScrambleOnChange 0x0004`
Obsolete flag. Provided for compatibility purposes only.

`#define memDebugModeValidateParams 0x0100`
: Force the heap library to thoroughly validate all parameters passed to the Memory Manager and Heap Manager functions. This validation includes pointers and memory chunk handles, so, for example, an attempt to resize a bad pointer can be detected.

Comments Use `MemDebugMode()` to obtain the current debug mode for the instance of the Heap Manager local to the calling process. Use `MemSetDebugMode()` to change the current debug mode.

Dynamic Heap Options

Purpose Pass these constants to `MemDynHeapOption()` to get or set various dynamic heap parameters at run time.

Declared In `MemoryMgr.h`

Constants

`#define memOptGetAbsMaxMemUsage 2`
: Retrieve the maximum amount of physical memory the dynamic heap is allowed to use.

`#define memOptGetAbsMinMemUsage 4`
: This option is not supported in Palm OS Cobalt.

`#define memOptGetForceMemReleaseThreshold 8`
: Retrieve the memory usage watermark above which all unused memory will be immediately released back to the operating system.

`#define memOptGetMaxUnusedMem 6`
: This option is not supported in Palm OS Cobalt.

`#define memOptSetAbsMaxMemUsage 1`
: Specify the maximum amount of physical memory the dynamic heap is allowed to use.

`#define memOptSetAbsMinMemUsage 3`
: This option is not supported in Palm OS Cobalt.

`#define memOptSetForceMemReleaseThreshold 7`
: Specify the memory usage watermark above which all unused memory will be immediately released back to the operating system. The default value is the size of the heap, so this feature is off by default.

`#define memOptSetMaxUnusedMem 5`
This option is not supported in Palm OS Cobalt.

Heap Flags

Purpose The set of flags that can be obtained for a heap using `MemHeapFlags()`.

Declared In `MemoryMgr.h`

Constants

`#define memHeapFlagReadOnly memHeapFlagROMBased`
The heap is read-only; it cannot be written to.

`#define memHeapFlagROMBased 0x0001`
The heap is located in ROM.

`#define memHeapFlagWritable 0x0002`
The heap can be written to.

Memory Manager Error Codes

Purpose Error codes returned by the various Memory Manager functions.

Declared In `MemoryMgr.h`

Constants

```
#define memErrAlreadyInitialized (memErrorClass |
  13)

#define memErrCardNotPresent (memErrorClass | 5)

#define memErrChunkLocked (memErrorClass | 1)

#define memErrChunkNotLocked (memErrorClass | 4)

#define memErrEndOfHeapReached (memErrorClass |
  15)

#define memErrFirst memErrChunkLocked
```

```
#define memErrHeapInvalid (memErrorClass | 14)

#define memErrInvalidParam (memErrorClass | 3)

#define memErrInvalidStoreHeader (memErrorClass |
  7)

#define memErrLast memErrEndOfHeapReached

#define memErrNoCardHeader (memErrorClass | 6)

#define memErrNoRAMOnDevice (memErrorClass | 10)

#define memErrNoStore (memErrorClass | 11)

#define memErrNotEnoughSpace (memErrorClass | 2)

#define memErrRAMOnlyDevice (memErrorClass | 8)

#define memErrROMOnlyDevice (memErrorClass | 12)

#define memErrWriteProtect (memErrorClass | 9)
```

LocalIDKind Enum

Purpose

Declared In `MemoryMgr.h`

Constants `memIDPtr`

`memIDHandle`

Memory Manager Functions and Macros

MemCmp Function

Purpose Compare two blocks of memory.

Declared In `MemoryMgr.h`

Prototype
```
int16_t MemCmp (const void *s1, const void *s2,
   int32_t numBytes)
```

Parameters

→ *`s1`*
Pointer to the first block of memory to be compared.

→ *`s2`*
Pointer to the second block of memory to be compared.

→ *`numBytes`*
Number of bytes to compare.

Returns Returns zero if the two blocks of memory match, a positive value if s1 > s2, and a negative value if s1 < s2.

Comments The two memory blocks are compared as a set of unsigned bytes.

MemDebugMode Function

Purpose Obtain the current debug mode for the instance of the Heap Manager local to the calling process.

Declared In `MemoryMgr.h`

Prototype `uint16_t MemDebugMode (void)`

Parameters None.

Returns Returns a set of debug flags. See "Debug Mode Flags" on page 266 for the set of flags that this function can return.

See Also `MemSetDebugMode()`

MemDynHeapGetInfo Function

Purpose Retrieve information about a dynamic heap.

Declared In `MemoryMgr.h`

Prototype
```
status_t MemDynHeapGetInfo
    (MemHeapInfoType *oInfo)
```

Parameters ← *`oInfo`*

Pointer to a structure that gets filled with information about the dynamic heap. See "MemHeapInfoType" on page 264.

Returns Always returns `errNone`.

Comments Your application must supply a `MemHeapInfoType` structure to this function. Upon return, the structure contains the following information:

- The size of the largest chunk that could be potentially allocated.
- The default alignment of memory chunks.
- The base address of the dynamic heap.
- The amount of virtual address space reserved for the heap
- The amount of physical memory that could be used to extend the pool of memory chunks.
- The amount of physical memory being used by the dynamic heap, and the amount that could be returned to the operating system.
- The number of chunks allocated from the heap, and the number of chunks in the heap that are free.
- The amount of memory used by chunks that are not free.
- The amount of uncommitted virtual address space reserved for chunks.
- The total number of bytes that could potentially be used to allocate chunks.
- The size of the largest memory block that could be allocated from the dynamic heap, and the size of the largest memory

block that could be allocated from the dynamic heap without using additional kernel memory.

See Also `MemDynHeapOption()`, `MemDynHeapReleaseUnused()`, `MemHeapDynamic()`

MemDynHeapOption Function

Purpose Allow the fine-tuning of various dynamic heap parameters at run time.

Declared In `MemoryMgr.h`

Prototype

```
uint32_t MemDynHeapOption (uint32_t cmd,
   uint32_t value)
```

Parameters

→ *`cmd`*
One of the commands listed under "Dynamic Heap Options" on page 267.

→ *`value`*
The value associated with the command, when using one of the option-setting commands. Ignored otherwise.

Returns Returns the current effective value of the specified dynamic heap option.

See Also `MemDynHeapGetInfo()`, `MemHeapDynamic()`

MemDynHeapReleaseUnused Function

Purpose Force the dynamic heap to release as much memory as it can back to the operating system.

Declared In `MemoryMgr.h`

Prototype `void MemDynHeapReleaseUnused (void)`

Parameters None.

Returns Nothing.

Comments The Heap Manager releases unused memory in page quantities. Any page in the address range controlled by the heap that does not contain allocated memory chunks or internal heap control structures could potentially be released back to the operating system. Applications should not assume that all pages occupied by the heap are always accessible; never attempt to access, for example, the area occupied by a chunk that was freed.

See Also `MemHeapDynamic()`

MemHandleDataStorage Function

Purpose Determine whether or not a chunk is located in a storage heap.

Declared In `MemoryMgr.h`

Prototype `Boolean MemHandleDataStorage (MemHandle h)`

Parameters → *h*
Chunk handle.

Returns Returns `true` if the specified chunk belongs to the storage area.

See Also `MemPtrDataStorage()`

MemHandleFree Function

Purpose Dispose of a memory chunk given its handle.

Declared In `MemoryMgr.h`

Prototype `status_t MemHandleFree (MemHandle h)`

Parameters → *h*
Chunk handle.

Returns Returns `errNone` if no error occurred. Returns `memErrInvalidParam` if the chunk could not, or should not, be freed.

Comments If the `memDebugModeFillFree` flag is set, the unused memory will be filled with a default value (currently, `0x55`).

If the supplied pointer indicates a chunk in a storage heap, the request is forwarded to the Data Manager.

NOTE: The Palm OS Cobalt Memory Manager uses virtual pages to hold handle tables, and they may not be returned to the kernel even if the chunks referenced by those handles are freed. In addition, the threshold of free memory that a heap can keep without returning the memory to the kernel impacts the amount of free memory reported after certain allocation and de-allocation operations. Because of this, if you allocate handles and pointers and then free them, the amount of memory reported as available after the series of operations may not be the same as that reported before.

See Also `MemDebugMode()`, `MemPtrFree()`, `MemHandleNew()`, `DmHandleFree()`

MemHandleHeapID Function

Purpose Get the ID of the heap that contains a given memory chunk referenced by its handle.

Declared In `MemoryMgr.h`

Prototype `uint16_t MemHandleHeapID (MemHandle h)`

Parameters → *h*

Chunk handle.

Returns Returns the ID of the heap containing the specified memory chunk, or `0xFFFF` if the specified pointer does not match any heap.

See Also `MemHeapID()`, `MemPtrHeapID()`

MemHandleLock Function

Purpose Lock a chunk and obtain a pointer to the chunk's data.

Declared In `MemoryMgr.h`

Prototype `MemPtr MemHandleLock (MemHandle h)`

Parameters → *h*

Chunk handle.

Returns Returns a pointer to the chunk's data, or `NULL` if an error.

Comments A `NULL` handle can safely be passed to this function; `NULL` will be returned.

If the supplied handle indicates a chunk in a storage heap, the request is forwarded to the Data Manager.

See Also `MemHandleUnlock()`, `DmHandleLock()`

MemHandleNew Function

Purpose Allocate a new movable chunk in the dynamic heap.

Declared In `MemoryMgr.h`

Prototype `MemHandle MemHandleNew (uint32_t *size*)`

Parameters → *`size`*
Size, in bytes, of the memory chunk to allocate.

Returns Returns the handle of the chunk, or `NULL` if the chunk couldn't be allocated.

Comments The handle returned by this function should not be interpreted by the application in any way. Memory handles should be used only in conjunction with the appropriate APIs.

See Also `MemHandleFree()`, `MemPtrNew()`

MemHandleResize Function

Purpose Resize a chunk referenced by a handle.

Declared In `MemoryMgr.h`

Prototype
```
status_t MemHandleResize (MemHandle h,
   uint32_t newSize)
```

Parameters → *`h`*
Chunk handle.

→ *`newSize`*
New size of the memory chunk. This value should be non-zero.

Returns Returns `errNone` if the chunk was successfully resized, or one of the following otherwise:

`memErrNotEnoughSpace`
There is not enough free memory to fulfill the request.

`memErrChunkLocked`
The given chunk cannot be resized.

`memErrInvalidParam`
One of the supplied arguments is invalid.

Comments This function may cause the unlocked chunk to be moved.

If the supplied handle indicates a chunk in a storage heap, the request is forwarded to the Data Manager.

See Also `MemHandleSize()`, `MemPtrResize()`, `DmHandleResize()`

MemHandleSetOwner Function

Purpose Set the owner ID of a chunk, given the chunk's handle.

Declared In `MemoryMgr.h`

Prototype

```
status_t MemHandleSetOwner (MemHandle h,
   uint16_t owner)
```

Parameters

→ *`h`*
Chunk handle.

→ *`owner`*
New owner ID of the chunk. Specify 0 to set the owner to the operating system. Only the lowest four bits are used.

Returns Returns `errNone` if the owner ID was set successfully, or `memErrInvalidParam` if an error occurred.

Comments The Heap Manager reserves owner ID 15 for internal usage. You cannot set a chunk's owner ID to 15 with this function.

See Also `MemPtrSetOwner()`

MemHandleSize Function

Purpose Get the size of a memory chunk referenced by a handle.

Declared In `MemoryMgr.h`

Prototype `uint32_t MemHandleSize (MemHandle h)`

Parameters → *h*
Chunk handle.

Returns Returns the size, in bytes, of the memory chunk referenced by the handle. Returns 0 if the size of the chunk is 0 or if an error occurred.

Comments If the supplied handle indicates a chunk in a storage heap, the request is forwarded to the Data Manager.

See Also `MemHandleResize()`, `MemPtrRealloc()`, `DmHandleSize()`

MemHandleUnlock Function

Purpose Unlock a movable memory chunk.

Declared In `MemoryMgr.h`

Prototype `status_t MemHandleUnlock (MemHandle h)`

Parameters → *h*
Chunk handle.

Returns Returns `errNone` if the chunk was unlocked, or `memErrInvalidParam` if an error occurred.

Comments If the supplied handle indicates a chunk in a storage heap, the request is forwarded to the Data Manager.

See Also `MemHandleLock()`, `MemPtrUnlock()`, `DmHandleUnlock()`

MemHeapCheck Function

Purpose Validate the internal structure of a given heap.

Declared In `MemoryMgr.h`

Prototype `status_t MemHeapCheck (uint16_t heapID)`

Parameters → *heapID*
ID of the heap to check.

Returns Returns `errNone` if the operation completed successfully, or one of the following otherwise:

`memErrInvalidParam`
: *`heapID`* is invalid.

`memErrInvalidHeap`
: Heap corruption was detected.

Comments This function can be used with any writable heap. If the calling process does not have write access to the heap, `errNone` is returned. This call is never forwarded to the Data Manager.

This function is called internally at appropriate times if the `MemDebugModeCheckOnChange` or `memDebugModeCheckOnAll` debug mode flags are set.

See Also `MemDebugMode()`, `MemHeapCompact()`

MemHeapCompact Function

Purpose Compact a heap.

Declared In `MemoryMgr.h`

Prototype `status_t MemHeapCompact (uint16_t` *`heapID`*`)`

Parameters → *`heapID`*
: ID of the heap to be compacted.

Returns Returns `errNone` if the operation completed successfully, or one of the following otherwise:

`memErrInvalidParam`
: *`heapID`* is invalid, or the heap specified by *`heapID`* is not writable.

`memErrNotEnoughSpace`
: There was not enough memory to complete the compaction.

Comments The calling process must have write permission to be able to compact the heap. If the calling process does not have write access to the heap, `errNone` is returned.

This call is never forwarded to the Data Manager.

See Also `MemHeapScramble()`

MemHeapDynamic Function

Purpose Determine whether or not the specified heap is the dynamic heap.

Declared In `MemoryMgr.h`

Prototype `Boolean MemHeapDynamic (uint16_t heapID)`

Parameters → *heapID*
ID of the heap.

Returns Returns `true` if the specified heap is the dynamic heap, `false` otherwise.

See Also `MemDynHeapGetInfo()`, `MemDynHeapOption()`, `MemDynHeapReleaseUnused()`, `MemHeapFlags()`

MemHeapFlags Function

Purpose Get the heap flags for a specified heap. These flags indicate whether or not the heap can be written to and whether or not the heap is located in ROM.

Declared In `MemoryMgr.h`

Prototype `uint16_t MemHeapFlags (uint16_t heapID)`

Parameters → *heapID*
ID of the heap.

Returns Returns the heap flags, or 0 if *heapID* is invalid. See "Heap Flags" on page 268 for the set of flags that can make up the returned value.

See Also `MemHeapDynamic()`

MemHeapFreeBytes Function

Purpose Get the total number of free bytes in a specified heap and the size of the largest free chunk in that heap.

Declared In `MemoryMgr.h`

Prototype
```
status_t MemHeapFreeBytes (uint16_t heapID,
   uint32_t *freeP, uint32_t *maxP)
```

Parameters → *heapID*
ID of the heap.

← *`freeP`*
The total number of bytes that are free in the heap.

← *`maxP`*
The size, in bytes, of the largest free chunk in the heap.

Returns Returns `errNone` if the operation completed successfully, or `memErrInvalidParam` if *`heapID`* is invalid.

Comments The size of the largest chunk returned by this call, in most cases, will be the size of the heap “wilderness” area: the area that is not backed up with physical memory. There is no guarantee that the returned amount actually can be allocated due to limits on physical memory imposed by resource bank and overall availability of free memory in the system.

See Also `MemHeapSize()`

MemHeapID Function

Purpose Get the ID for a heap, given its index.

Declared In `MemoryMgr.h`

Prototype `uint16_t MemHeapID (uint16_t` *`heapIndex`*`)`

Parameters → *`heapIndex`*
Heap index.

Returns Returns the heap ID.

Comments Index 0 refers to the dynamic heap. Index 1 refers to the storage area. Index 2 refers to ROM.

See Also `MemHandleHeapID()`, `MemPtrHeapID()`

MemHeapScramble Function

Purpose Scramble a heap, moving each of the heap's movable chunks. This function can be useful when debugging.

Declared In `MemoryMgr.h`

Prototype `status_t MemHeapScramble (uint16_t heapID)`

Parameters → *`heapID`*
ID of the heap to be scrambled.

Returns Returns `errNone` if the operation completed successfully, or one of the following otherwise:

`memErrInvalidParam`
`heapID` is invalid, or the heap specified by *`heapID`* is not writable.

`memErrNotEnoughSpace`
There was not enough memory to scramble the heap.

Comments The calling process must have write permission to be able to scramble the heap. If the calling process does not have write access to the heap, `errNone` is returned.

This call is never forwarded to the Data Manager.

See Also `MemHeapCompact()`

MemHeapSize Function

Purpose Get the maximum number of bytes that the heap can manage or request from the kernel.

Declared In `MemoryMgr.h`

Prototype `uint32_t MemHeapSize (uint16_t heapID)`

Parameters → *`heapID`*
ID of the heap.

Returns Returns the maximum size, in bytes, of the specified heap, or 0 if *`heapID`* is invalid.

Comments The value returned by this call represents the maximum amount possible. Not all of this memory is necessarily available.

See Also `MemHeapFreeBytes()`

MemMove Function

Purpose Move memory.

Declared In `MemoryMgr.h`

Prototype
```
status_t MemMove (void *dstP, const void *sP,
   int32_t numBytes)
```

Parameters

← *`dstP`*
Pointer to the destination.

→ *`sP`*
Pointer to the source.

→ *`numBytes`*
Number of bytes to move.

Returns Always returns `errNone`.

Comments This function properly handles overlapping ranges.

MemNumHeaps Function

Purpose Get the number of available heaps in both ROM and RAM.

Declared In `MemoryMgr.h`

Prototype `uint16_t MemNumHeaps (void)`

Parameters None.

Returns The number of heaps. This value is always 3, since the system has three heaps: the dynamic heap, the storage area, and ROM.

See Also `MemHandleHeapID()`, `MemPtrHeapID()`, `MemNumRAMHeaps()`

MemNumRAMHeaps Function

Purpose Get the number of available RAM heaps.

Declared In `MemoryMgr.h`

Prototype `uint16_t MemNumRAMHeaps (void)`

Parameters None.

Returns The number of heaps. This value is always 2, since the system has two RAM heaps: the dynamic heap, and the non-secure RAM storage heap.

See Also `MemHandleHeapID()`, `MemPtrHeapID()`, `MemNumHeaps()`

MemPtrDataStorage Function

Purpose Determine whether or not a chunk is located in the storage heap.

Declared In `MemoryMgr.h`

Prototype `Boolean MemPtrDataStorage (MemPtr p)`

Parameters → *p*
Pointer to the chunk.

Returns Returns `true` if the specified chunk belongs to the storage area.

Comments This function checks whether or not the given pointer falls within the address range occupied by the heap located in the storage area.

See Also `MemHandleDataStorage()`

MemPtrFree Macro

Purpose Dispose of a memory chunk referenced by the given pointer.

Declared In `MemoryMgr.h`

Prototype `#define MemPtrFree (p)`

Parameters → *p*
Pointer to the memory chunk to be freed.

Returns Returns `errNone` if no error occurred. Returns `memErrInvalidParam` if the chunk could not, or should not, be freed.

Comments If the `memDebugModeFillFree` flag is set, the unused memory will be filled with a default value (currently, `0x55`).

If the supplied pointer indicates a chunk in a storage heap, the request is forwarded to the Data Manager.

NOTE: The Palm OS Cobalt Memory Manager uses virtual pages to hold handle tables, and they may not be returned to the kernel even if the chunks referenced by those handles are freed. In addition, the threshold of free memory that a heap can keep without returning the memory to the kernel impacts the amount of free memory reported after certain allocation and de-allocation operations. Because of this, if you allocate handles and pointers and then free them, the amount of memory reported as available after the series of operations may not be the same as that reported before.

See Also `MemDebugMode()`, `MemHandleFree()`, `MemPtrNew()`, `DmHandleFree()`

MemPtrHeapID Function

Purpose Get the ID of the heap that contains a given memory chunk referenced by a pointer.

Declared In `MemoryMgr.h`

Prototype `uint16_t MemPtrHeapID (MemPtr p)`

Parameters → *p*
Pointer to the chunk.

Returns Returns the ID of the heap containing the specified memory chunk, or `0xFFFF` if the specified pointer does not match any heap.

See Also `MemHandleHeapID()`, `MemHeapID()`

MemPtrNew Function

Purpose Allocate a new memory chunk from the dynamic heap.

Declared In `MemoryMgr.h`

Prototype `MemPtr MemPtrNew (uint32_t size)`

Parameters → *size*
The desired size of the chunk.

Returns Returns a pointer to a newly allocated chunk if successful, or `NULL` if the Memory Manager was unable to allocate a memory chunk of the requested size.

Comments This function allocates a non-movable chunk in the dynamic heap and returns a pointer to that chunk. Applications can use this call to allocate dynamic memory. User processes should use this call as a primary dynamic memory allocator.

See Also `MemHandleNew()`, `MemPtrFree()`

MemPtrRealloc Function

Purpose Change the size of a non-movable chunk referenced by a pointer.

Declared In `MemoryMgr.h`

Prototype
```
MemPtr MemPtrRealloc (MemPtr ptr,
   uint32_t newSize)
```

Parameters

→ *`ptr`*
Pointer to the memory chunk to be reallocated.

→ *`newSize`*
New size, in bytes, of the chunk.

Returns Returns a pointer to the reallocated chunk, or `NULL` if the chunk couldn't be resized as requested.

Comments The semantic of this call resembles the standard C library function `realloc`. The contents of the chunk will be unchanged up to the lesser of the new and old size. If *`ptr`* is `NULL`, this function behaves like `MemPtrNew()`. If *`newSize`* is 0 and *`ptr`* is not `NULL`, the memory chunk is freed and `NULL` is returned. `MemPtrRealloc` significantly simplifies the management of variable-length memory chunks, so this call is recommended over `MemPtrResize()`.

Only non-movable chunks can be reallocated using this call.

See Also `MemHandleResize()`, `DmPtrResize()`

MemPtrRecoverHandle Function

Purpose Recover the handle of a memory chunk referenced by the given pointer to its data.

Declared In `MemoryMgr.h`

Prototype `MemHandle MemPtrRecoverHandle (MemPtr p)`

Parameters → *`p`*
Pointer to a memory chunk.

Returns Returns the handle of the memory chunk, or `NULL` if an error occurred.

Comments For memory chunks in the dynamic heap, the given pointer will be converted to a handle and returned as a result. For memory chunks in a storage heap, the call is forwarded to the Data Manager.

See Also `DmRecoverHandle()`

MemPtrResize Function

Purpose Resize a memory chunk referenced by a pointer.

Declared In `MemoryMgr.h`

Prototype
```
status_t MemPtrResize (MemPtr p,
   uint32_t newSize)
```

Parameters → *`p`*
Pointer to the memory chunk to be resized.

→ *`newSize`*
New desired size of the memory chunk, in bytes.

Returns Returns `errNone` if the chunk was successfully resized, or one of the following otherwise:

`memErrNotEnoughSpace`
There is not enough memory to fulfill the request.

`memErrChunkLocked`
The given chunk cannot be resized in place.

`memErrInvalidParam`
One of the arguments is invalid, the chunk does not exist, or the chunk should not be resized.

Comments Call this function to resize a locked chunk. This function is always successful when shrinking the size of a chunk. When growing a chunk, it attempts to use free space immediately following the chunk, and returns `memErrChunkLocked` if the resize fails.

For non-movable chunks in the dynamic heap, consider using `MemPtrRealloc()`. In most cases, that function is more convenient.

If the supplied pointer indicates a chunk in a storage heap, the request is forwarded to the Data Manager.

See Also `MemHandleResize()`, `MemPtrNew()`, `DmPtrResize()`

MemPtrSetOwner Function

Purpose Set the owner ID of a chunk referenced by a pointer.

Declared In `MemoryMgr.h`

Prototype
```
status_t MemPtrSetOwner (MemPtr p,
   uint16_t owner)
```

Parameters

→ *p*
Chunk pointer.

→ *owner*
New owner ID of the chunk. Specify 0 to set the owner to the operating system. Only the lowest four bits are used.

Returns Returns `errNone` if the owner ID was set successfully, or `memErrInvalidParam` if an error occurred.

Comments The Heap Manager reserves owner ID 15 for internal usage. You cannot set a chunk's owner ID to 15 with this function.

See Also `MemHandleSetOwner()`

MemPtrSize Function

Purpose Get the size of a memory chunk referenced by a pointer.

Declared In `MemoryMgr.h`

Prototype `uint32_t MemPtrSize (MemPtr p)`

Parameters → *p*
Pointer to a memory chunk.

Returns The size of the chunk, in bytes, or 0 if an error occurred.

Comments The value returned represents the size of the “Data” portion of the memory chunk that is equal to the value that was specified when it was allocated or resized.

If the supplied pointer indicates a chunk in a storage heap, the request is forwarded to the Data Manager.

See Also `MemPtrNew()`, `MemPtrResize()`, `DmPtrSize()`

MemPtrUnlock Function

Purpose Unlock a chunk, given a pointer to the chunk.

Declared In `MemoryMgr.h`

Prototype `status_t MemPtrUnlock (MemPtr p)`

Parameters → *p*
Pointer to the chunk to be unlocked.

Returns Returns `errNone` if the chunk was unlocked, or `memErrInvalidParam` if an error occurred.

Comments If the supplied pointer indicates a chunk in a storage heap, the request is forwarded to the Data Manager.

See Also `MemHandleUnlock()`, `DmPtrUnlock()`

MemSet Function

Purpose Set a memory range to a specified value.

Declared In `MemoryMgr.h`

Prototype
```
status_t MemSet (void *dstP, int32_t numBytes,
   uint8_t value)
```

Parameters ← *dstP*
Pointer to the beginning of the memory range to be set.

→ *`numBytes`*
Number of bytes to be set.

→ *`value`*
Value to which each of the bytes in the specified range are set.

Returns Always returns `errNone`.

MemSetDebugMode Function

Purpose Set the debugging mode for the instance of the Heap Manager local to the calling process.

Declared In `MemoryMgr.h`

Prototype `status_t MemSetDebugMode (uint16_t` *`flags`*`)`

Parameters → *`flags`*
Use the logical OR operator (|) to provide any combination of the flags listed in "Debug Mode Flags" on page 266.

Returns Returns `errNone` if the debug mode flags were set successfully, or `memErrHeapInvalid` if an invalid heap was detected.

Comments When using the `memDebugModeFillFree` debug flag, note that only memory that is accessible will be filled. The first 32 bits of free chunk data are reserved for internal use and will never be filled.

When working with the storage heap you should try to always use functions provided by the Data Manager. The `MemDebugModeNoDmCalls` debug flag helps you to track down "leftover" Memory Manager calls that operate on the storage heap. These calls can then be converted into Data Manager calls.

See Also `MemDebugMode()`

7

Schema Databases

This chapter describes the schema database APIs: those structures, constants, and functions that operate on schema databases. This chapter is divided into the following sections:

The header file `SchemaDatabases.h` declares the API that this chapter describes.

For more information on Palm OS® databases, see Chapter 2, "Palm OS Databases," on page 11.

Schema Databases Structures and Types

DbColumnPropertySpecType Struct

Purpose Used in conjunction with `DbGetColumnPropertyValues()` to specify column properties for selective value retrieval.

Declared In `SchemaDatabases.h`

Prototype

```
typedef struct {
   uint32_t columnID;
   DbSchemaColumnProperty propertyID;
   uint8_t padding[3];
} DbColumnPropertySpecType,
*DbColumnPropertySpecPtr
```

Fields `columnID`

The ID of the column for which the property is being retrieved.

`propertyID`
: The ID of the property being retrieved. See `DbSchemaColumnProperty`.

`padding`
: Padding bytes used for structure alignment purposes.

DbColumnPropertyValueType Struct

Purpose Container that identifies a single column property and contains its value.

Declared In `SchemaDatabases.h`

Prototype

```
typedef struct {
   uint32_t columnID;
   uint32_t dataSize;
   void *data;
   status_t errCode;
   DbSchemaColumnProperty propertyID;
   uint8_t padding[3];
} DbColumnPropertyValueType,
*DbColumnPropertyValuePtr
```

Fields

`columnID`
: The ID of the column for which the property is being retrieved or set.

`dataSize`
: The size, in bytes, of the property value.

`data`
: The property value.

`errCode`
: Set by the Data Manager to `errNone` if the property value was set or retrieved successfully, or one of the Data Manager error codes otherwise.

`propertyID`
: The ID of the property being retrieved or set.

`padding`
: Padding bytes used for structure alignment purposes only.

Comments You work with an array of these structures when getting or setting column property values with

`DbGetAllColumnPropertyValues()`, `DbGetColumnPropertyValues()`, and `DbSetColumnPropertyValues()`.

DbMatchModeType Typedef

Purpose Define how a row's category membership should match a supplied set of categories.

Declared In `SchemaDatabases.h`

Prototype `typedef uint32_t DbMatchModeType`

Constants `#define DbMatchAll ((DbMatchModeType)2)`
(AND) Match rows for which membership includes all of the specified categories, including rows with additional category membership.

`#define DbMatchAny ((DbMatchModeType)1)`
(OR) Match rows for which membership includes any of the specified categories.

`#define DbMatchExact ((DbMatchModeType)3)`
Match rows for which membership exactly matches the specified categories.

DbSchemaColumnData Typedef

Purpose Generic type for any kind of column data.

Declared In `SchemaDatabases.h`

Prototype `typedef void DbSchemaColumnData;`

Fields None.

Comments The `DbSchemaColumnValueType` structure uses this data type for the column's data.

DbSchemaColumnDefnType Struct

Purpose Defines a single table column.

Declared In `SchemaDatabases.h`

Prototype

```
typedef struct {
   uint32_t id;
   uint32_t maxSize;
   char name[dbDBNameLength];
   DbSchemaColumnType type;
   uint8_t attrib;
   uint16_t reserved;
   status_t errCode;
} DbSchemaColumnDefnType, *DbSchemaColumnDefnPtr
```

Fields

`id`
: User-defined column identifier.

`maxSize`
: Size specification for the column data. For variable-length string vectors, it specifies the size upper-bound and for fixed-length strings, the actual size. For vectors, it specifies the upper-bound in terms of byte count. For all other types, the actual size of the type.

`name`
: User-defined column name.

`type`
: The column type. See the definition of `DbSchemaColumnType` for a list of supported column types.

`attrib`
: Column attributes. See "Table Column Attributes" on page 301 for a list of supported column attributes.

`reserved`
: Reserved for future use.

`errCode`
: Set by the Data Manager to an error code in the course of a value retrieval operation. `errNone` represents a no-error condition.

Comments You work with these structures both singly and in arrays when adding columns and getting column definitions with `DbAddColumn()`, `DbGetAllColumnDefinitions()`, and

`DbGetColumnDefinitions()`. A table definition contains an array of these structures; see "DbTableDefinitionType" on page 299.

DbSchemaColumnProperty Typedef

Purpose Container for a column property's type.

Declared In `SchemaDatabases.h`

Prototype `typedef uint8_t DbSchemaColumnProperty`

Constants

```
#define dbColumnAttribProperty
  ((DbSchemaColumnProperty)0x04)
```
The column's attributes.

```
#define dbColumnDatatypeProperty
  ((DbSchemaColumnProperty)0x02)
```
The column's data type.

```
#define dbColumnNameProperty
  ((DbSchemaColumnProperty)0x01)
```
The column's name.

```
#define dbColumnSizeProperty
  ((DbSchemaColumnProperty)0x03)
```
The column's size.

Comments Pass these values directly when setting or getting a single table column property value with `DbSetColumnPropertyValue()` or `DbGetColumnPropertyValue()`, or when removing a column property with `DbRemoveColumnProperty()`. When getting or setting multiple property values, you use these values in conjunction with one or more `DbColumnPropertyValueType` structures.

DbSchemaColumnType Typedef

Purpose Contains a value identifying the type of a table column.

Declared In `SchemaDatabases.h`

Prototype `typedef uint8_t DbSchemaColumnType`

Constants

`#define dbBlob ((DbSchemaColumnType)0x11)`
A blob. This data type supports offset-based reads and writes.

`#define dbBoolean ((DbSchemaColumnType)0x0B)`
A Boolean.

`#define dbChar ((DbSchemaColumnType)0x0F)`
A char.

`#define dbDate ((DbSchemaColumnType)0x0D)`
A date.

`#define dbDateTime ((DbSchemaColumnType)0x0C)`
A date and time, not including seconds.

`#define dbDateTimeSecs ((DbSchemaColumnType)0x12)`
A date and time, including seconds.

`#define dbDouble ((DbSchemaColumnType)0x0A)`
A double.

`#define dbFloat ((DbSchemaColumnType)0x09)`
A float.

`#define dbInt16 ((DbSchemaColumnType)0x06)`
A signed 16-bit integer.

`#define dbInt32 ((DbSchemaColumnType)0x07)`
A signed 32-bit integer.

`#define dbInt64 ((DbSchemaColumnType)0x08)`
A signed 64-bit integer.

`#define dbInt8 ((DbSchemaColumnType)0x05)`
A signed 8-bit integer.

`#define dbStringVector ((DbSchemaColumnType)0xC0)`
A string vector.

`#define dbTime ((DbSchemaColumnType)0x0E)`
A time.

`#define dbUInt16 ((DbSchemaColumnType)0x02)`
An unsigned 16-bit integer.

`#define dbUInt32 ((DbSchemaColumnType)0x03)`
An unsigned 32-bit integer.

`#define dbUInt64 ((DbSchemaColumnType)0x04)`
An unsigned 64-bit integer.

`#define dbUInt8 ((DbSchemaColumnType)0x01)`
An unsigned 8-bit integer.

`#define dbVarChar ((DbSchemaColumnType)0x10)`
A VarChar. This data type supports offset-based reads and writes.

`#define dbVector ((DbSchemaColumnType)0x80)`
A vector. This data type supports offset-based reads and writes.

Comments These constants are used when adding columns to a table or getting table column definitions.

DbSchemaColumnValueType Struct

Purpose Identifies a table column and acts as a container for the column's data. You use this structure primarily when reading and writing multiple column values in a database row.

Declared In `SchemaDatabases.h`

Prototype

```
typedef struct {
   DbSchemaColumnData *data;
   uint32_t dataSize;
   uint32_t columnID;
   uint32_t columnIndex;
   status_t errCode;
   uint32_t reserved;
} DbSchemaColumnValueType,
*DbSchemaColumnValuePtr
```

Fields `data`
The column data.

`dataSize`
: The size, in bytes, of the column data being read or written. For variable-length string types, it specifies the actual size to be read or written. For vectors, it specifies the actual byte count to be read or written. When writing, `*data` must, at a minimum, have storage corresponding to `dataSize`.

`columnID`
: The column ID.

`columnIndex`
: The column index. This field is only used when reading column data.

`errCode`
: Set by the Data Manager to an error code in the course of a value retrieval operation. `errNone` represents a no-error condition.

`reserved`
: Reserved for future use.

Comments Use this structure when reading or writing multiple data columns in a single operation with the following functions:

- `DbCopyColumnValues()`
- `DbGetAllColumnValues()`
- `DbGetColumnValues()`
- `DbWriteColumnValues()`

You also use this data structure with `DbInsertRow()`.

DbShareModeType Typedef

Purpose Container for the share mode type, which controls how others can access a database that your application has opened using either `DbOpenDatabase()` or `DbOpenDatabaseByName()`.

Declared In `SchemaDatabases.h`

Prototype `typedef uint16_t DbShareModeType`

Constants `#define dbShareNone ((DbShareModeType)0x0000)`
: While the database is open, don't let anyone else open it.

`#define dbShareRead ((DbShareModeType)0x0001)`
While the database is open, others can open it in read-only mode.

`#define dbShareReadWrite ((DbShareModeType)0x0002)`
While the database is open, others can open it in read-only, read-write, or write-only mode.

DbTableDefinitionType Struct

Purpose Defines a database table. This structure contains the table's name, and acts as a container for an array of `DbSchemaColumnDefnType` structures, each element of which defines an individual column.

Declared In `SchemaDatabases.h`

Prototype

```
typedef struct {
   char name[dbDBNameLength];
   uint32_t numColumns;
   DbSchemaColumnDefnType *columnListP;
} DbTableDefinitionType
```

Fields

`name`
Table name.

`numColumns`
Number of columns in the table, which is also the number of elements in the *`columnListP`* array.

`columnListP`
Pointer to the first of a set of data structures that each define a single table column.

Comments You use this structure when creating a database with either `DbCreateDatabase()` or `DbCreateSecureDatabase()`, when adding a new table to a database (with `DbAddTable()`) and when querying a database table for schema information (`DbGetTableSchema()`).

Schema Databases Constants

Schema Database Row Attributes

Purpose Define the set of attributes that a row can have. Use `DbGetRowAttr()` to obtain a row's attributes.

Declared In `DataMgr.h`

Constants

`#define dbRecAttrArchive 0x01`
: The row is marked for archiving: it is treated like a deleted row, but the chunk is not freed and the row ID is preserved so that upon the next HotSync operation the desktop computer saves the row data before it permanently removes the row entry and data from the Palm Powered™ handheld.

`#define dbRecAttrDelete 0x80`
: The row has been deleted.

`#define dbRecAttrReadOnly 0x02`
: The row is read-only, and cannot be written to. Note that the Data Manager does not place any semantics on the read-only attribute. It is up to the application to enforce the read-only semantics.

`#define dbRecAttrSecret 0x10`
: The row is private.

```
#define dbAllRecAttrs (dbRecAttrDelete |
  dbRecAttrSecret | dbRecAttrArchive |
  dbRecAttrReadOnly)
```
The complete set of schema database row attributes.

```
#define dbSysOnlyRecAttrs (dbRecAttrDelete |
  dbRecAttrArchive)
```
System-only attributes. These attributes are maintained by the operating system and cannot be set with `DbSetRowAttr()`.

Table Column Attributes

Purpose Identify the various attributes of a table column.

Declared In `DataMgr.h`

Constants

`#define dbSchemaColDynamic 0x01`
The column was added after the table was created.

`#define dbSchemaColNonSyncable 0x02`
The column's data won't be synchronized. Modifications made to a "non-syncable" column's data don't change the modification state for the row, and thus by themselves don't cause the row to be synchronized during a HotSync operation.

`#define dbSchemaColWritable 0x04`
The column's data can be modified. Writable columns are relevant for read-only rows and are required for sharing.

```
#define dbAllSchemaColAttrs (dbSchemaColDynamic |
  dbSchemaColNonSyncable | dbSchemaColWritable)
```
The complete set of table column attributes.

Comments These constants are used when adding columns to a table or getting table column definitions. See "DbSchemaColumnDefnType" on page 294 for more information.

Schema Database Access Rule Action Types

Purpose Database actions that can have access rules set in a secure database.

Declared In `DataMgr.h`

Constants

`#define dbActionBackup ((AzmActionType)0x00000008)`
Database backup is permitted.

`#define dbActionDelete ((AzmActionType)0x00000004)`
Database contents can be deleted.

```
#define dbActionEditSchema
  ((AzmActionType)0x00000020)
```
Database schemas can be altered.

`#define dbActionRead ((AzmActionType)0x00000001)`
The database can be read.

`#define dbActionRestore ((AzmActionType)0x00000010)`
Database restore is permitted.

`#define dbActionWrite ((AzmActionType)0x00000002)`
The database can be written to.

Comments Use a combination of these values (or'd together) to create the `action` parameter you supply to `AzmAddRule()`.

Cursor Open Flags

Purpose Flags used to specify how a database cursor is created. Supply any combination of these (OR'd together) to `DbCursorOpen()` or `DbCursorOpenWithCategory()`; in most cases you supply none of these flags (that is, you supply a `flags` value of zero).

Declared In `SchemaDatabases.h`

Constants `#define dbCursorEnableCaching 0x00010000`
Enable the caching of row data locally in the cursor.

`#define dbCursorIncludeDeleted 0x00000001`
The cursor should contain rows that are marked as deleted or archived.

`#define dbCursorOnlyDeleted 0x00000002`
The cursor should contain *only* those rows that are marked as deleted or archived.

`#define dbCursorOnlySecret 0x00000004`
The cursor should contain *only* those rows that are marked as secret.

`#define dbCursorSortByCategory 0x10000000`
Sort rows by category. Rows with multiple categories appear in the cursor multiple times.

Miscellaneous Schema Database Constants

Purpose The header file `SchemaDatabases.h` also declares these constants.

Declared In `SchemaDatabases.h`

Constants

`#define dbColumnPropertyUpperBound ((DbSchemaColumnProperty)0x0A)`
: Identifies the upper bound of the range of built-in property type IDs.

`#define DbCursorBOFPos 0xFFFFFFFF`
: Cursor row position signifying BOF (Beginning Of File).

`#define dbCursorEOFPos 0xFFFFFFFE`
: Cursor row position signifying EOF (End Of File).

`#define dbDBNameLength 32`
: Maximum length, including the `NUL` terminator, of a schema database name.

`#define dbInvalidCursorID 0x0`
: Cursor ID returned from `DbCursorOpen()` or `DbCursorOpenWithCategory()` if the open failed.

`#define dbInvalidRowID dbInvalidCursorID`
: Row ID returned from `DbCursorGetCurrentRowID()`, `DbCursorGetPositionForRowID()`, or `DbInsertRow()` if the operation failed.

`#define DbMaxRecordCategories 255`
: Maximum number of categories to which a row can be assigned.

`#define dbMaxRowIndex 0x00FFFFFEL`
: Highest possible row index.

DbFetchType Enum

Purpose Specifies how the cursor is to be repositioned when using `DbCursorMove()`.

Declared In `SchemaDatabases.h`

Constants

`dbFetchRelative`
: Moves the cursor forward by the specified number of rows if the offset is positive, or backward by the specified number of rows if the offset is negative.

`dbFetchAbsolute`
: Moves the cursor onto the row with the specified index. The macro `DbCursorSetAbsolutePosition()` calls `DbCursorMove()` with a fetch type of `dbFetchAbsolute`.

`dbFetchNext`
: Moves the cursor forward one row. The macro `DbCursorMoveNext()` calls `DbCursorMove()` with a fetch type of `dbFetchNext`. Note that the *`offset`* parameter to `DbCursorMove()` is ignored when the fetch type is `dbFetchNext`.

`dbFetchPrior`
: Moves the cursor backward one row. The macro `DbCursorMovePrev()` calls `DbCursorMove()` with a fetch type of `dbFetchPrior`. Note that the *`offset`* parameter to `DbCursorMove()` is ignored when the fetch type is `dbFetchPrior`.

`dbFetchFirst`
: Moves the cursor onto the first row. The macro `DbCursorMoveFirst()` calls `DbCursorMove()` with a fetch type of `dbFetchFirst`.

`dbFetchLast`
: Moves the cursor onto the last row. The macro `DbCursorMoveLast()` calls `DbCursorMove()` with a fetch type of `dbFetchLast`.

`dbFetchRowID`
: Moves the cursor onto the row with the specified row ID. The macro `DbCursorMoveToRowID()` calls `DbCursorMove()` with a fetch type of `dbFetchRowID`.

Comments Any attempt to move the current row position beyond the set of rows in the cursor results in an error. `DbCursorMove()` returns `dmErrCursorBOF` if you attempt to move before the first row in the cursor, and `dmErrCursorEOF` if you attempt to move beyond the last row in the cursor. These conditions can also be detected with the use of the `DbCursorIsBOF()` and `DbCursorIsEOF()` functions.

Schema Databases Functions and Macros

DbAddCategory Function

Purpose Make the specified row a member of one or more additional categories.

Declared In `DataMgr.h`

Prototype

```
status_t DbAddCategory (DmOpenRef dbRef,
   uint32_t rowID, uint32_t numToAdd,
   const CategoryID categoryIDs[])
```

Parameters

→ *dbRef*
`DmOpenRef` to an open database.

→ *rowID*
Row ID or cursor ID identifying the row to which categories are to be added.

→ *numToAdd*
Number of categories in the *categoryIDs* array.

→ *categoryIDs*
Array of category IDs.

Returns Returns `errNone` if no error, or one of the following if an error occurs:

`dmErrInvalidParam`
dbRef doesn't reference an open database, the specified row or cursor ID is not valid, or *numToAdd* is nonzero and *categoryIDs* is `NULL`.

`dmErrNotSchemaDatabase`
The specified database is not a schema database.

`dmErrReadOnly`
The database is not open for write access.

`dmErrIndexOutOfRange`
: The specified row or cursor ID doesn't reference a row within the table.

`dmErrRecordDeleted`
: The specified row is marked as deleted.

`dmErrRecordBusy`
: The specified row is in use and cannot be updated.

`dmErrMemError`
: A memory error occurred.

`dmErrInvalidCategory`
: The allowed number of categories has been exceeded, or a category ID doesn't correspond to a defined category.

Comments The database must be opened with write access.

The category IDs passed through the *`categoryIDs`* parameter must be valid category IDs. If any of the array values is not a valid category ID, this function returns `dmErrInvalidCategory`.

If a given category ID value appears more than once in the *`categoryIDs`* array, the category membership is only added once. If the row already has membership in a category specified in the *`categoryIDs`* array, the array value is ignored and the row remains a member of that category.

See Also `DbRemoveCategory()`, `DbSetCategory()`

DbAddColumn Function

Purpose Adds a column to a database table.

Declared In `SchemaDatabases.h`

Prototype

```
status_t DbAddColumn (DmOpenRef dbRef,
   const char *table,
   const DbSchemaColumnDefnType *addColumnP)
```

Parameters

→ *`dbRef`*
: `DmOpenRef` to an open database.

→ *`table`*
: Name of the table to which the column is to be added.

→ *`addColumnP`*
Pointer to a `DbSchemaColumnDefnType` structure defining the column to be added.

Returns Returns `errNone` if successful, or one of the following if an error occurred:

`dmErrInvalidParam`
`dbRef` doesn't reference an open database, *`table`* is `NULL`, or *`addColumnP`* is `NULL`.

`dmErrNotSchemaDatabase`
The specified database is not a schema database.

`dmErrReadOnly`
The database is not open for write access.

`dmErrInvalidColType`
The specified column type is not a valid column type.

`dmErrAccessDenied`
The database is a secure database and you don't have permission to edit its schemas.

`dmErrInvalidColSpec`
At least one of the specified column attributes is not a valid column attribute.

`dmErrInvalidColumnName`
The supplied column name is not a valid column name.

`dmErrInvalidVectorType`
The column is a vector column but the column type isn't appropriate for a vector column.

`dmErrInvalidSizeSpec`
The column is a vector column but the column size is zero.

`dmErrInvalidTableName`
The supplied table name doesn't identify a table in the specified database.

`dmErrColumnDefinitionsLocked`
The table's column definitions are locked.

`dmErrColumnIDAlreadyExists`
A column with the specified ID already exists.

`dmErrColumnNameAlreadyExists`
The table already contains a column with the specified name.

`dmErrMemError`
A memory error occurred.

Comments The database must be opened in write mode.

See Also `DbGetAllColumnDefinitions()`, `DbGetColumnDefinitions()`

DbAddSortIndex Function

Purpose Adds a new sort index to a database.

Declared In `SchemaDatabases.h`

Prototype
```
status_t DbAddSortIndex (DmOpenRef dbRef,
   const char *orderBy)
```

Parameters

→ *`dbRef`*
`DmOpenRef` to an open database.

→ *`orderBy`*
The sort index, which identifies both the table containing the rows to select from and the manner in which the cursor's rows should be sorted. See "The SELECT Statement" on page 37 for the format of this parameter.

Returns Returns `errNone` if successful, or one of the following if an error occurred:

`dmErrInvalidParam`
`dbRef` doesn't reference an open database.

`dmErrNotSchemaDatabase`
The specified database is not a schema database.

`dmErrReadOnly`
The database is not open for write access.

`dmErrAccessDenied`
You do not have authorization to modify the database.

`dmErrSQLParseError`
The specified table name or the sort information specified in the sort index is invalid.

`dmErrInvalidTableName`
The specified table doesn't exist within the database.

`dmErrInvalidSortDefn`
: The sort index contains no column IDs, or all of the columns in the sort index aren't of the same type.

`dmErrInvalidColumnID`
: One or more of the specified column IDs doesn't correspond to a column in the specified table.

`dmErrAlreadyExists`
: The specified sort index already exists.

`dmErrMemError`
: A memory error occurred.

Comments While sorting is enabled, the operating system keeps schema databases sorted according to each of the database's sort indices. This function adds a new sort index to a schema database. When the new sort index is added, the database is immediately sorted according to the new sort index.

Before you can open a cursor with a given sort index, the sort index must have already been added to the database.

The *`orderBy`* parameter is an SQL statement of the form described under "The SELECT Statement" on page 37. The optional WHERE clause allows you to filter the rows to be included in the cursor. The column specified in the WHERE clause can only be one of the following types:

- `dbDateTimeSecs`
- `dbBoolean`
- `dbVarChar`

With `dbVarChar` columns, the operator (*op*) can be "`LIKE`" (and the argument must be a string); this uses `TxtFindString()` to identify all rows where the supplied string is found in the row.

See Also `DbCursorOpen()`, `DbCursorOpenWithCategory()`, `DbHasSortIndex()`, `DbRemoveSortIndex()`

DbAddTable Function

Purpose Adds a table to an existing database.

Declared In `SchemaDatabases.h`

Prototype
```
status_t DbAddTable (DmOpenRef dbRef,
   const DbTableDefinitionType *schemaP)
```

Parameters

→ *`dbRef`*
: `DmOpenRef` to an open database.

→ *`schemaP`*
: Pointer to a structure that represents the table to be added.

Returns Returns `errNone` if successful, or one of the following if an error occurred:

`dmErrInvalidParam`
: *`dbRef`* doesn't reference an open database, or *`schemaP`* is `NULL`.

`dmErrNotSchemaDatabase`
: The database referenced by *`dbRef`* isn't a schema database.

`dmErrReadOnly`
: The database is read-only.

`dmErrInvalidSchemaDefn`
: The supplied `DbTableDefinitionType` structure is invalid.

`dmErrTableNameAlreadyExists`
: The database already contains a table with the specified name.

`dmErrColumnIDAlreadyExists`
: The table definition contains multiple columns with the same ID.

`dmErrColumnNameAlreadyExists`
: The table definition contains multiple columns with the same name.

`dmErrInvalidColType`
: A column data type is invalid.

`dmErrAccessDenied`
: You do not have authorization to modify the database.

Comments The database must be opened in write mode.

See Also `DbHasTable()`, `DbRemoveTable()`

DbArchiveRow Function

Purpose Mark a row as archived. This function leaves the row's data intact.

Declared In `SchemaDatabases.h`

Prototype

```
status_t DbArchiveRow (DmOpenRef dbRef,
   uint32_t rowID)
```

Parameters

→ *dbRef*
: `DmOpenRef` to an open database.

→ *rowID*
: Row ID or cursor ID identifying the row to be archived.

Returns Returns `errNone` if successful, or one of the following if an error occurred:

`dmErrReadOnly`
: You've attempted to write to or modify a database that is open in read-only mode.

`dmErrIndexOutOfRange`
: The specified index is out of range.

`dmErrRecordArchived`
: The function requires that the row not be archived, but it is.

`dmErrRecordDeleted`
: The row has been deleted.

Comments When a row is archived, the archive bit is set but the data chunks are not freed and the row ID is preserved. The next time the handheld is synchronized with the desktop computer, a conduit can save the row data on the desktop and then remove the row entry and data from the handheld.

See Also `DbCursorArchiveAllRows()`, `DbDeleteRow()`, `DbRemoveRow()`

DbCloseDatabase Function

Purpose Close a schema database.

Declared In `SchemaDatabases.h`

Prototype
```
status_t DbCloseDatabase (DmOpenRef dbRef)
```

Parameters

→ *`dbRef`*
`DmOpenRef` to an open database.

Returns Returns `errNone` if successful, or `dmErrInvalidParam` if the *`dbRef`* parameter doesn't indicate an open schema database.

Comments This function doesn't unlock any rows that were left locked. Applications should not leave rows locked when closing a schema database.

See Also `DbOpenDatabase()`, `DbOpenDatabaseByName()`

DbCopyColumnValue Function

Purpose Obtains the value of a single schema database column for a specified row.

Declared In `SchemaDatabases.h`

Prototype
```
status_t DbCopyColumnValue (DmOpenRef dbRef,
   uint32_t rowID, uint32_t columnID,
   uint32_t offset, void *valueP,
   uint32_t *valueSizeP)
```

Parameters

→ *`dbRef`*
`DmOpenRef` to an open database.

→ *`rowID`*
Row ID or cursor ID identifying the row for which column values are being retrieved.

→ *`columnID`*
ID of the column being retrieved.

→ *`offset`*
Column value offset from which the data is retrieved. This parameter is treated as a byte offset. See the Comments section, below, for more information.

↔ *valueP*
Pointer to a pre-allocated buffer into which the row’s column value is copied, or `NULL` to determine how large the buffer should be.

↔ *valueSizeP*
Size of the *valueP* buffer.

Returns Returns `errNone` if successful, or one of the following if an error occurred:

`dmErrInvalidParam`
rowID is not a row or cursor ID, or *valueSizeP* is `NULL`.

`dmErrCursorBOF`
The supplied cursor ID is BOF.

`dmErrCursorEOF`
The supplied cursor ID is EOF.

`dmErrUniqueIDNotFound`
The supplied cursor ID represents an invalid row.

`dmErrNotSchemaDatabase`
The specified database is not a schema database.

`dmErrReadOnly`
The database is not open for write access.

`dmErrUniqueIDNotFound`
The supplied row or cursor ID doesn’t correspond to a row within the database.

`dmErrRecordDeleted`
The specified row is marked as deleted.

`dmErrInvalidColSpec`
There are no columns defined for the specified table.

`dmErrInvalidColumnID`
The supplied column ID is invalid.

`dmErrNoColumnData`
The specified row has no data for the column.

`dmErrReadOutOfBounds`
The specified offset exceeds the bounds of the column.

`dmErrBufferNotLargeEnough`
: The supplied buffer isn't large enough to contain the column value.

`dmErrMemError`
: A memory error occurred.

Comments This function returns a copy of the column data. Offset-based reads are not supported for fixed-length column data types; the *`offset`* parameter is ignored for those data types. The list of column data types supporting offset-based reads are:

- `VarChar`
- `Blob`
- `Vector`

If *`valueP`* is `NULL`, this function returns the actual size needed to hold the column data through *`valueSizeP`*.

See Also `DbCopyColumnValues()`, `DbGetColumnValue()`, `DbWriteColumnValue()`

DbCopyColumnValues Function

Purpose Obtains the value of one or more schema database columns for a specified row.

Declared In `SchemaDatabases.h`

Prototype

```
status_t DbCopyColumnValues (DmOpenRef dbRef,
   uint32_t rowID, uint32_t numColumns,
   DbSchemaColumnValueType *columnValuesP)
```

Parameters

→ *`dbRef`*
: `DmOpenRef` to an open database.

→ *`rowID`*
: Row ID or cursor ID identifying the row for which column values are being retrieved.

→ *`numColumns`*
: Number of elements in the *`columnValuesP`* array.

↔ *`columnValuesP`*
: Pointer to a pre-allocated array of `DbSchemaColumnValueType` structures. Prior to calling

this function, the data field of each structure must be initialized with a pointer to a buffer of appropriate size for the column, or set to `NULL`, which results in the actual size of the column data being returned in the `actualDataSize` field of the structure.

Returns Returns `errNone` if successful, or one of the following if an error occurred:

`dmErrInvalidParam`
: *`rowID`* is not a row or cursor ID, *`dbRef`* doesn't reference an open database, *`numColumns`* is zero, or *`columnValuesP`* is `NULL`.

`dmErrCursorBOF`
: The supplied cursor ID is BOF.

`dmErrCursorEOF`
: The supplied cursor ID is EOF.

`dmErrUniqueIDNotFound`
: The supplied cursor ID represents an invalid row.

`dmErrNotSchemaDatabase`
: The specified database is not a schema database.

`dmErrReadOnly`
: The database is not open for write access.

`dmErrUniqueIDNotFound`
: The supplied row or cursor ID doesn't correspond to a row within the database.

`dmErrRecordDeleted`
: The specified row is marked as deleted.

`dmErrInvalidColSpec`
: There are no columns defined for the specified table.

`dmErrInvalidColumnID`
: The one or more of the specified column IDs is invalid.

`dmErrNoColumnData`
: The specified row has no data.

`dmErrBufferNotLargeEnough`
: At least one of the supplied buffers isn't large enough to contain the corresponding column value.

`dmErrMemError`
A memory error occurred.

See Also `DbCopyColumnValue()`, `DbGetColumnValues()`, `DbWriteColumnValues()`

DbCreateDatabase Function

Purpose Creates a new schema database.

Declared In `SchemaDatabases.h`

Prototype
```
status_t DbCreateDatabase (const char *name,
   uint32_t creator, uint32_t type,
   uint32_t numTables,
   const DbTableDefinitionType schemaListP[],
   DatabaseID *dbIDP)
```

Parameters

→ *`name`*
Name of the new database. The name should be up to 32 ASCII bytes long, including the `NULL` terminator, as specified by `dmDBNameLength`. The name should be constructed only of 7-bit ASCII characters (`0x20` through `0x7E`).

→ *`creator`*
Database creator ID.

→ *`type`*
Database type.

→ *`numTables`*
Number of elements in *`schemaListP`*. This parameter can be zero, which creates a new database with no tables defined.

→ *`schemaListP`*
Array of structures. Each element defines the schema for the newly-created database table.

← *`dbIDP`*
ID of the newly-created database. Pass this ID to `DbOpenDatabase()` when opening the database.

Returns Returns `errNone` if successful, or one of the following if an error occurred:

`dmErrInvalidDatabaseName`
: The specified database name is nonexistent, exceeds `dmDBNameLength`, or is otherwise invalid.

`dmErrInvalidSchemaDefn`
: *`schemaListP`* is `NULL`, no table name was supplied, one or more column names are missing.

`dmErrTableNameAlreadyExists`
: One of the supplied table names occurs in more than one *`schemaListP`* entry.

`dmErrColumnIDAlreadyExists`
: One of the supplied column IDs is already defined for this database.

`dmErrColumnNameAlreadyExists`
: One of the supplied column names is already defined for this database.

`dmErrInvalidColType`
: One of the supplied column types is not a valid column type.

`dmErrInvalidVectorType`
: A one of the supplied vector column types isn't a valid vector column type.

`dmErrInvalidSizeSpec`
: At least one of the vector column sizes is zero.

`dmErrInvalidColSpec`
: One of the supplied column attributes is not a valid column attribute.

`dmErrInvalidColumnName`
: One or more table or column names was invalid.

`dmErrAccessDenied`
: You don't have permission to create a database of this type.

`dmErrAlreadyExists`
: Another database with this name already exists.

`dmErrMemError`
: A memory error occurred. Sufficient memory must be available to create a new database.

`memErrNotEnoughSpace`
: A memory error occurred.

Comments Prior to calling this function, the database must not already exist. Sufficient memory must be available to create a new database. If *`numTables`* is nonzero, the supplied `DbTableDefinitionType` structures must have been previously initialized.

See Also `DbCreateSecureDatabase()`, `DbCreateSecureDatabaseFromImage()`, `DbOpenDatabase()`

DbCreateSecureDatabase Function

Purpose Create a new secure schema database.

Declared In `SchemaDatabases.h`

Prototype

```
status_t DbCreateSecureDatabase
   (const char *name, uint32_t creator,
   uint32_t type, uint32_t numSchemas,
   const DbTableDefinitionType schemaList[],
   AzmRuleSetType *ruleset, DatabaseID *id)
```

Parameters

→ *`name`*
Name of the new database. The name should be up to 32 ASCII bytes long, including the `NULL` terminator, as specified by `dmDBNameLength`. The name should be constructed only of 7-bit ASCII characters (`0x20` through `0x7E`).

→ *`creator`*
Database creator ID.

→ *`type`*
Database type.

→ *`numSchemas`*
Number of elements in *`schemaList`*. This parameter can be zero, which creates a secure database with no tables defined.

→ *`schemaList`*
Array of structures. Each element defines the schema for the newly-created database table.

← *`ruleset`*
Handle to the database's access rules.

← *`id`*
ID of the newly-created database. Pass this ID to `DbOpenDatabase()` when opening the database.

Returns Returns `errNone` if the database was successfully created, or one of the following if there was an error:

`dmErrInvalidDatabaseName`
: The specified database name is nonexistent, exceeds `dmDBNameLength`, or is otherwise invalid.

`dmErrInvalidSchemaDefn`
: *`schemaListP`* is `NULL`, no table name was supplied, one or more column names are missing.

`dmErrTableNameAlreadyExists`
: One of the supplied table names occurs in more than one *`schemaListP`* entry.

`dmErrColumnIDAlreadyExists`
: One of the supplied column IDs is already defined for this database.

`dmErrColumnNameAlreadyExists`
: One of the supplied column names is already defined for this database.

`dmErrInvalidColType`
: One of the supplied column types is not a valid column type.

`dmErrInvalidVectorType`
: A one of the supplied vector column types isn’t a valid vector column type.

`dmErrInvalidSizeSpec`
: At least one of the vector column sizes is zero.

`dmErrInvalidColSpec`
: One of the supplied column attributes is not a valid column attribute.

`dmErrInvalidColumnName`
: One or more column names was invalid.

`dmErrInvalidTableName`
: One or more table names was invalid.

`dmErrAccessDenied`
: You don’t have permission to create a database of this type.

`dmErrAlreadyExists`
: Another database with this name already exists.

`dmErrMemError`
: A memory error occurred. Sufficient memory must be available to create a new database.

`memErrNotEnoughSpace`
: A memory error occurred.

`azmErrOutOfMemory`
: A memory error occurred.

Comments Prior to calling this function, the database must not already exist. Sufficient memory must be available to create a new database. If *`numSchemas`* is nonzero, the supplied `DbTableDefinitionType` structures must have been previously initialized.

Once the database is created, it is initially protected with all actions (Read, Write, Delete, Schema Edit, Backup, and Restore) denied. Before using the database you must specify access rules for the Read, Write, and Delete actions using Authorization Manager and Authentication Manager functions (see *Exploring Palm OS: Security and Cryptography* for documentation on these functions). Until access rules are specified, all access to the database is denied.

See Also `DbCreateDatabase()`, `DbCreateSecureDatabaseFromImage()`, `DbOpenDatabase()`

DbCreateSecureDatabaseFromImage Function

Purpose Create a secure schema database from a single resource that contains an image of the database.

Declared In `SchemaDatabases.h`

Prototype

```
status_t DbCreateSecureDatabaseFromImage
   (const void *bufferP, DatabaseID *pDbID,
   AzmRuleSetType *pRuleSet)
```

Parameters

→ *`bufferP`*
: Pointer to a locked resource containing the database image.

← *`pDbID`*
: Pointer to a variable that receives the ID of the newly-created database, or `NULL` if the ID isn't needed.

← *pRuleSet*
Pointer to the Authorization Manager rule set for the newly-created secure database.

Returns Returns `errNone` if the database was successfully created. Otherwise, this function returns an error code such as (but not limited to) the following:

`dmErrInvalidParam`
bufferP is `NULL` or *pRuleSet* is `NULL`.

`dmErrCorruptDatabase`
The format of the supplied database image isn't recognizable as a schema database.

`dmErrMemError`
A memory error occurred.

Comments This function is typically used by applications to install a default database.

See Also `DbCreateDatabase()`, `DbCreateSecureDatabase()`, `DbOpenDatabase()`

DbCursorArchiveAllRows Function

Purpose Mark all rows in the cursor for archiving.

Declared In `SchemaDatabases.h`

Prototype
```
status_t DbCursorArchiveAllRows
   (uint32_t cursorID)
```

Parameters → *cursorID*
ID of a valid cursor, as returned from `DbCursorOpen()` or `DbCursorOpenWithCategory()`.

Returns Returns `errNone` if the operation completed successfully, or one of the following otherwise:

`dmErrInvalidParam`
The specified cursor ID isn't valid.

`dmErrRecordBusy`
One of the rows is in use and cannot be updated.

`dmErrReadOnly`
: You've attempted to write to or modify a database that is open in read-only mode.

Comments When a row is archived, the archive bit is set but the data chunks are not freed and the row ID is preserved. The next time the handheld is synchronized with the desktop computer, a conduit can save the row data on the desktop and then remove the row entry and data from the handheld.

See Also `DbArchiveRow()`, `DbCursorDeleteAllRows()`, `DbCursorRemoveAllRows()`

DbCursorBindData Function

Purpose Bind a variable to a cursor column.

Declared In `SchemaDatabases.h`

Prototype

```
status_t DbCursorBindData (uint32_t cursorID,
   uint32_t columnID, void *dataBufferP,
   uint32_t dataBufferLength,
   uint32_t *dataSizeP, status_t *errCodeP)
```

Parameters

→ *cursorID*
: ID of a valid cursor, as returned from `DbCursorOpen()` or `DbCursorOpenWithCategory()`.

→ *columnID*
: ID of the column to be bound.

← *dataBufferP*
: Pointer to a buffer that receives the bound field data.

→ *dataBufferLength*
: Size, in bytes, of the data buffer specified in *dataBufferP*.

← *dataSizeP*
: The size of the data written to the data buffer.

← *errCodeP*
: An error code that is updated whenever the data buffer is updated.

Returns Returns `errNone` if the data buffer is successfully bound to the column, or one of the following otherwise:

`dmErrInvalidParam`
: The function received an invalid parameter.

`dmErrMemError`
: A memory error occurred.

Comments When a variable is bound to column, that variable is automatically updated to hold the field value of the cursor's current row. Using the `DbCursorMove...` functions and macros to change the current row in the cursor automatically updates any bound variables.

When the bound variable is updated, `*dataSizeP` is set to the size of the data stored in the data buffer. This is useful for columns of varying length types (`VarChar` and `Blob`), but is not needed for fixed length types. The error code is also set each time the variable is updated, indicating success (`errNone`), no data for that column (`dmErrNoColumnData`), or some other failure error code.

See Also `DbCursorBindDataWithOffset()`, `DbCursorMove()`, `DbCursorOpen()`, `DbCursorOpenWithCategory()`, `DbCursorUpdate()`

DbCursorBindDataWithOffset Function

Purpose Bind a variable to a cursor column, offset by a specified amount.

Declared In `SchemaDatabases.h`

Prototype
```
status_t DbCursorBindDataWithOffset
   (uint32_t cursorID, uint32_t columnID,
   void *dataBufferP, uint32_t dataBufferLength,
   uint32_t *dataSizeP, uint32_t fieldDataOffset,
   status_t *errCodeP)
```

Parameters

→ *`cursorID`*
: ID of a valid cursor, as returned from `DbCursorOpen()` or `DbCursorOpenWithCategory()`.

→ *`columnID`*
: ID of the column to be bound.

← *`dataBufferP`*
: Pointer to a buffer that receives the bound field data.

→ *`dataBufferLength`*
: Size, in bytes of the data buffer specified in *`dataBufferP`*.

← *dataSizeP*
The size of the data written to the data buffer.

→ *fieldDataOffset*
Byte offset into the column.

← *errCodeP*
An error code that is updated whenever the data buffer is updated.

Returns Returns `errNone` if the data buffer is successfully bound to the column, or one of the following otherwise:

`dmErrInvalidParam`
The function received an invalid parameter.

`dmErrMemError`
A memory error occurred.

Comments This function is similar to `DbCursorBindData()`, but adds an extra parameter to allow you to specify an offset into the database field data. The data copied to the buffer is taken from the database field at the specified offset. This allows you to bind a subset of the field data to a variable.

See Also `DbCursorMove()`, `DbCursorOpen()`, `DbCursorOpenWithCategory()`, `DbCursorUpdate()`

DbCursorClose Function

Purpose Free all resources associated with a cursor.

Declared In `SchemaDatabases.h`

Prototype `status_t DbCursorClose (uint32_t` *`cursorID`*`)`

Parameters → *cursorID*
ID of a valid cursor, as returned from `DbCursorOpen()` or `DbCursorOpenWithCategory()`.

Returns Returns `errNone` if the resources were successfully freed, or one of the following otherwise:

`dmErrInvalidParam`
The supplied cursor ID is invalid.

`dmErrMemError`
A memory error occurred.

Comments When a cursor is no longer needed, call `DbCursorClose()` to free all of the resources associated with the cursor.

See Also `DbCursorOpen()`, `DbCursorOpenWithCategory()`

DbCursorDeleteAllRows Function

Purpose Mark all rows in the cursor as deleted.

Declared In `SchemaDatabases.h`

Prototype

```
status_t DbCursorDeleteAllRows
    (uint32_t cursorID)
```

Parameters

→ *`cursorID`*
ID of a valid cursor, as returned from `DbCursorOpen()` or `DbCursorOpenWithCategory()`.

Returns Returns `errNone` if the operation completed successfully, or one of the following otherwise:

`dmErrInvalidParam`
The specified cursor ID isn't valid.

`dmErrRecordBusy`
One of the rows is in use and cannot be updated.

`dmErrReadOnly`
You've attempted to write to or modify a database that is open in read-only mode.

Comments For each row in the cursor, this function deletes the row's chunk from the database but leaves the row entry in the header and marks the row as deleted. During the next HotSync operation, a conduit can save the row data on the desktop and then remove the row entries in the header that are marked as deleted.

See Also `DbCursorArchiveAllRows()`, `DbCursorRemoveAllRows()`, `DbDeleteRow()`

DbCursorFlushCache Function

Purpose Flush the contents of the cursor cache. This function should only be called for cursors that were created with caching enabled.

Declared In `SchemaDatabases.h`

Prototype `status_t DbCursorFlushCache (uint32_t cursorID)`

Parameters → *`cursorID`*
ID of a valid cursor, as returned from `DbCursorOpen()` or `DbCursorOpenWithCategory()`.

Returns Returns `errNone` if the operation completed successfully, or one of the following otherwise:

`dmErrInvalidParam`
The specified cursor ID is not valid.

`dmErrAccessDenied`
The specified cursor wasn't created with caching enabled. That is, the `dbCursorEnableCaching` flag was not specified when the cursor was created.

`dmErrRecordBusy`
The specified cursor is in use.

DbCursorGetCurrentPosition Function

Purpose Get the index of the cursor's current row.

Declared In `SchemaDatabases.h`

Prototype
```
status_t DbCursorGetCurrentPosition
    (uint32_t cursorID, uint32_t *position)
```

Parameters → *`cursorID`*
ID of a valid cursor, as returned from `DbCursorOpen()` or `DbCursorOpenWithCategory()`.

← *`position`*
The row index of the current row within the cursor.

Returns Returns `errNone` if *`*position`* was set to a valid row index, or one of the following otherwise:

`dmErrInvalidParam`
The specified cursor ID is not valid.

`dmErrCursorBOF`
The current position is before the first cursor row.

`dmErrCursorEOF`
The current position is after the last cursor row.

Comments The first row within a cursor has an index value of 1.

See Also `DbCursorGetCurrentRowID()`, `DbCursorGetPositionForRowID()`, `DbCursorGetRowIDForPosition()`

DbCursorGetCurrentRowID Function

Purpose Get the row ID of the cursor's current row.

Declared In `SchemaDatabases.h`

Prototype
```
status_t DbCursorGetCurrentRowID
    (uint32_t cursorID, uint32_t *rowIDP)
```

Parameters

→ *`cursorID`*
ID of a valid cursor, as returned from `DbCursorOpen()` or `DbCursorOpenWithCategory()`.

← *`rowIDP`*
Pointer to a variable that receives the row ID. If the cursor isn't currently positioned at a valid row, `*`*`rowIDP`* is set to `dbInvalidRowID`.

Returns Returns `errNone` if the operation completed successfully, or one of the following otherwise:

`dmErrInvalidParam`
`dbRef` doesn't reference an open database, or the specified cursor ID is not valid.

`dmErrCursorBOF`
The current position is before the first cursor row.

`dmErrCursorEOF`
The current position is after the last cursor row.

`dmErrUniqueIDNotFound`
: The current row's ID is invalid.

See Also `DbCursorGetCurrentPosition()`, `DbCursorGetPositionForRowID()`, `DbCursorGetRowIDForPosition()`, `DbGetTableForRow()`

DbCursorGetPositionForRowID Function

Purpose Get the index of a specified row within the cursor.

Declared In `SchemaDatabases.h`

Prototype
```
status_t DbCursorGetPositionForRowID
   (uint32_t cursorID, uint32_t rowID,
   uint32_t *positionP)
```

Parameters

→ *`cursorID`*
: ID of a valid cursor, as returned from `DbCursorOpen()` or `DbCursorOpenWithCategory()`.

→ *`rowID`*
: ID of a row within the cursor.

← *`positionP`*
: The index of the specified row within the cursor, or 0 if an error occurred.

Returns Returns `errNone` if the operation completed successfully, or one of the following otherwise:

`dmErrInvalidParam`
: The specified cursor ID is not valid, the specified row ID isn't a valid row ID, or *`positionP`* is `NULL`.

`dmErrCantFind`
: The specified row ID doesn't match any of the cursor's rows.

Comments The first row within a cursor has an index value of 1.

See Also `DbCursorGetCurrentPosition()`, `DbCursorGetCurrentRowID()`, `DbCursorGetRowIDForPosition()`

DbCursorGetRowCount Function

Purpose Get the total number of rows in the cursor.

Declared In `SchemaDatabases.h`

Prototype
```
uint32_t DbCursorGetRowCount (uint32_t cursorID)
```

Parameters

→ *cursorID*
ID of a valid cursor, as returned from `DbCursorOpen()` or `DbCursorOpenWithCategory()`.

Returns Returns the number of rows in the cursor.

DbCursorGetRowIDForPosition Function

Purpose Get a row's ID given its index.

Declared In `SchemaDatabases.h`

Prototype
```
status_t DbCursorGetRowIDForPosition
   (uint32_t cursorID, uint32_t position,
   uint32_t *rowIDP)
```

Parameters

→ *cursorID*
ID of a valid cursor, as returned from `DbCursorOpen()` or `DbCursorOpenWithCategory()`.

→ *position*
Index of the row for which the ID is to be retrieved.

← *rowIDP*
The row's ID. If row ID cannot be determined, `*`*rowIDP* is set to `dbInvalidRowID`.

Returns Returns `errNone` if the ID was successfully retrieved, or one of the following if an error occurred:

`dmErrInvalidParam`
The specified cursor ID is not valid, the specified position doesn't indicate a valid row within the cursor, or *rowIDP* is `NULL`.

`dmErrRecordDeleted`
The row at the specified position is marked for deletion.

Comments The first row within a cursor has an index value of 1.

See Also `DbCursorGetCurrentPosition()`, `DbCursorGetCurrentRowID()`, `DbCursorGetPositionForRowID()`

DbCursorIsBOF Function

Purpose Determine if the cursor's BOF (beginning of file) property is true.

Declared In `SchemaDatabases.h`

Prototype `Boolean DbCursorIsBOF (uint32_t cursorID)`

Parameters → *cursorID*
ID of a valid cursor, as returned from `DbCursorOpen()` or `DbCursorOpenWithCategory()`.

Returns Returns `true` if the cursor is at BOF, `false` otherwise.

Comments BOF is the position immediately *before* the first row in the cursor. Attempting to move before the first row in the cursor sets BOF to `true` and returns a `dmErrCursorBOF`. If BOF is `true`, moving to the next row moves to the first row in the cursor.

See Also `DbCursorIsEOF()`, `DbCursorMove()`, `DbCursorMoveFirst()`, `DbCursorMoveNext()`

DbCursorIsDeleted Function

Purpose Determine if the cursor's current row is marked for deletion.

Declared In `SchemaDatabases.h`

Prototype `Boolean DbCursorIsDeleted (uint32_t cursorID)`

Parameters → *cursorID*
ID of a valid cursor, as returned from `DbCursorOpen()` or `DbCursorOpenWithCategory()`.

Returns Returns `true` if the current row is marked for deletion, `false` otherwise. Note that this function returns `false` if the supplied cursor ID isn't valid, or if the cursor's current position doesn't represent a valid row (for instance, if the current position is at BOF).

See Also `DbArchiveRow()`, `DbDeleteRow()`

DbCursorIsEOF Function

Purpose Determine whether the cursor's EOF (end of file) property is true.

Declared In `SchemaDatabases.h`

Prototype `Boolean DbCursorIsEOF (uint32_t` *`cursorID`*`)`

Parameters → *`cursorID`*
ID of a valid cursor, as returned from `DbCursorOpen()` or `DbCursorOpenWithCategory()`.

Returns Returns `true` if the cursor is at EOF, `false` otherwise.

Comments EOF is the position immediately *after* the last row in the cursor. Attempting to move past the last row in the cursor sets EOF (end of file) to `true` and returns a `dmErrCursorEOF`. If EOF is `true`, moving to the previous row moves to the last row in the cursor.

See Also `DbCursorIsBOF()`, `DbCursorMove()`, `DbCursorMoveLast()`, `DbCursorMovePrev()`

DbCursorMove Function

Purpose Move a cursor's current row position.

Declared In `SchemaDatabases.h`

Prototype
```
status_t DbCursorMove (uint32_t cursorID,
   int32_t offset, DbFetchType fetchType)
```

Parameters → *`cursorID`*
ID of a valid cursor, as returned from `DbCursorOpen()` or `DbCursorOpenWithCategory()`.

→ *`offset`*
Number of rows to move the current row selector. Negative numbers move backward.

→ *`fetchType`*
One of the values defined by the `DbFetchType` enum specifying how the cursor is to move (forward one row, backward a specified number of rows, to an absolute position, etc.).

Returns Returns `errNone` if the current row position was moved to a valid row within the cursor, or one of the following otherwise:

`dmErrInvalidParam`
: The specified cursor ID is invalid.

`dmErrCursorBOF`
: An attempt was made to move to a position before the first row in the cursor.

`dmErrCursorEOF`
: An attempt was made to move to a position after the last row in the cursor.

Comments When *`fetchType`* is `dbFetchRelative`, positive values move the current row position forward, while negative values move the current row position backward. Attempting to move before the first row in the cursor, or attempting to move past the last row in the cursor generates an error, and the cursor's BOF or EOF property, as appropriate, is set.

When moving through the cursor, note that rows that were modified are not moved to their new sort position until `DbCursorRequery()` is called. Similarly, any new rows are not available to the cursor until `DbCursorRequery()` is called.

Upon successful completion of the move, any bound variables are updated with corresponding field values for the new current row.

See Also `DbCursorMoveFirst()`, `DbCursorMoveLast()`, `DbCursorMoveNext()`, `DbCursorMovePrev()`, `DbCursorMoveToRowID()`, `DbCursorSetAbsolutePosition()`

DbCursorMoveFirst Macro

Purpose Set the current row position of the cursor to the first row in the cursor.

Declared In `SchemaDatabases.h`

Prototype `#define DbCursorMoveFirst (`*`i`*`)`

Parameters → *`i`*
: ID of a valid cursor, as returned from `DbCursorOpen()` or `DbCursorOpenWithCategory()`.

Returns Returns `errNone` if the current row position was moved to a valid row within the cursor, or one of the following otherwise:

`dmErrInvalidParam`
: The specified cursor ID is invalid.

`dmErrCursorEOF`
: The cursor contains no rows.

Comments Upon successful completion of the move, any bound variables are updated with corresponding field values for the new current row.

See Also `DbCursorMove()`

DbCursorMoveLast Macro

Purpose Set the current row position of the cursor to the last row in the cursor.

Declared In `SchemaDatabases.h`

Prototype `#define DbCursorMoveLast (`*`i`*`)`

Parameters → *`i`*
: ID of a valid cursor, as returned from `DbCursorOpen()` or `DbCursorOpenWithCategory()`.

Returns Returns `errNone` if the current row position was moved to a valid row within the cursor, or one of the following otherwise:

`dmErrInvalidParam`
: The specified cursor ID is invalid.

`dmErrCursorBOF`
: The cursor contains no rows.

Comments Upon successful completion of the move, any bound variables are updated with corresponding field values for the new current row.

See Also `DbCursorMove()`

DbCursorMoveNext Macro

Purpose Move the cursor's current row position forward to the next row in the cursor.

Declared In `SchemaDatabases.h`

Prototype `#define DbCursorMoveNext (i)`

Parameters → *i*

ID of a valid cursor, as returned from `DbCursorOpen()` or `DbCursorOpenWithCategory()`.

Returns Returns `errNone` if the current row position was moved to a valid row within the cursor, or one of the following otherwise:

`dmErrInvalidParam`
The specified cursor ID is invalid.

`dmErrCursorEOF`
An attempt was made to move to a position after the last row in the cursor.

Comments An attempt to move past the last row in the cursor generates a `dmErrCursorEOF` error and sets the cursor's EOF property.

When moving through the cursor, note that rows that were modified are not moved to their new sort position until `DbCursorRequery()` is called. Similarly, any new rows are not available to the cursor until `DbCursorRequery()` is called.

Upon successful completion of the move, any bound variables are updated with corresponding field values for the new current row.

See Also `DbCursorMove()`

DbCursorMovePrev Macro

Purpose Move the cursor's current row position backward to the previous row in the cursor.

Declared In `SchemaDatabases.h`

Prototype `#define DbCursorMovePrev (i)`

Parameters → *i*

ID of a valid cursor, as returned from `DbCursorOpen()` or `DbCursorOpenWithCategory()`.

Returns Returns `errNone` if the current row position was moved to a valid row within the cursor, or one of the following otherwise:

`dmErrInvalidParam`
The specified cursor ID is invalid.

`dmErrCursorBOF`
An attempt was made to move to a position before the first row in the cursor.

Comments An attempt to move before the first row in the cursor generates a `dmErrCursorBOF` error and sets the cursor's BOF property.

When moving through the cursor, note that rows that were modified are not moved to their new sort position until `DbCursorRequery()` is called. Similarly, any new rows are not available to the cursor until `DbCursorRequery()` is called.

Upon successful completion of the move, any bound variables are updated with corresponding field values for the new current row.

See Also `DbCursorMove()`

DbCursorRelocateRow Function

Purpose Relocate a row within a cursor that was opened using the default sort index.

Declared In `SchemaDatabases.h`

Prototype
```
status_t DbCursorRelocateRow (uint32_t cursorID,
    uint32_t from, uint32_t to)
```

Parameters

→ *`cursorID`*
ID of a valid cursor, as returned from `DbCursorOpen()` or `DbCursorOpenWithCategory()`.

→ *`from`*
The index of the row to be moved.

→ *`to`*
The index of the position to which the row is to be moved.

Returns Returns `errNone` if the current row position was moved to a valid row within the cursor, or one of the following otherwise:

`dmErrInvalidParam`
: The specified cursor ID is invalid, or the cursor's sort index is not the default sort index.

`dmErrIndexOutOfRange`
: Either *`from`* or *`to`* exceeds the number of rows in the cursor.

Comments This function can only be used with cursors opened using the default sort index (that is, a cursor opened without an ORDER BY clause). It allows you to "manually" rearrange the order of the rows in the cursor.

If the row being moved is the current row, the cursor is updated so that the current row position is set to the new location of the moved row.

Cursor row positions are one-based. That is the first row in the cursor has an index value of 1. The last row in the cursor has an index value of `DbCursorGetRowCount()`.

See Also `DbCursorMove()`

DbCursorMoveToRowID Macro

Purpose Position the cursor at the row with the specified row ID.

Declared In `SchemaDatabases.h`

Prototype `#define DbCursorMoveToRowID (`*`i`*`, `*`r`*`)`

Parameters

→ *`i`*
: ID of a valid cursor, as returned from `DbCursorOpen()` or `DbCursorOpenWithCategory()`.

→ *`r`*
: The ID of the row to which the cursor is to be positioned.

Returns Returns `errNone` if the current row position was changed to a valid row within the cursor, or one of the following otherwise:

`dmErrInvalidParam`
: The specified cursor ID is invalid.

Comments Upon successful completion, any bound variables are updated with corresponding field values for the new current row.

See Also `DbCursorMove()`, `DbCursorSetAbsolutePosition()`

DbCursorOpen Function

Purpose Creates and opens a cursor containing all rows in the specified table that conform to a specified set of flags, ordered as specified. No filtering of rows based upon category membership is performed.

Declared In `SchemaDatabases.h`

Prototype

```
status_t DbCursorOpen (DmOpenRef dbRef,
    const char *sql, uint32_t flags,
    uint32_t *cursorID)
```

Parameters

→ *`dbRef`*
: `DmOpenRef` to an open database.

→ *`sql`*
: A sort index identifying both the table containing the rows to select from and the manner in which the cursor's rows should be sorted. *The sort index must have already been added to the table prior to its use here*; see "The SELECT Statement" on page 37 for the format of this parameter.

→ *`flags`*
: Zero or more flags (OR'd together) that specify how the cursor is to be opened. See "Cursor Open Flags" on page 302 for the set of flags defined for this operation.

← *`cursorID`*
: The ID of the newly-opened cursor. If there was an error opening the cursor, `*cursorID` is set to `dbInvalidCursorID`.

Returns Returns `errNone` if the cursor was successfully opened, or one of the following otherwise:

`dmErrInvalidParam`
: *`dbRef`* doesn't reference an open database, *`sql`* is `NULL`, or *`cursorID`* is `NULL`.

`dmErrInvalidSortIndex`
: One of the sort IDs specified in the supplied SQL isn't valid for the specified database table.

`dmErrMemError`
: The operation couldn't be completed due to insufficient memory.

`dmErrNotSchemaDatabase`
The database specified by *`dbRef`* isn't a schema database.

`dmErrSQLParseError`
The SQL specified in the *`sql`* parameter is invalid.

`dmErrCursorEOF`
The cursor was successfully created but the table contains no rows that match the specified criteria.

Comments If the ORDER BY clause is omitted (that is, if the SQL string consists solely of the table name, and perhaps a WHERE clause) the cursor rows are not sorted. Such a cursor is said to be opened using the **default sort index**.

When a cursor is no longer needed, call `DbCursorClose()` to free all resources associated with the cursor.

See Also `DbCursorClose()`, `DbCursorOpenWithCategory()`

DbCursorOpenWithCategory Function

Purpose Creates and opens a cursor containing all rows in the specified table that conform to a specified set of flags, ordered as specified. Rows are filtered based upon category membership.

Declared In `SchemaDatabases.h`

Prototype

```
status_t DbCursorOpenWithCategory
   (DmOpenRef dbRef, const char *sql,
   uint32_t flags, uint32_t numCategories,
   const CategoryID categoryIDs[],
   DbMatchModeType matchMode, uint32_t *cursorID)
```

Parameters

→ *`dbRef`*
`DmOpenRef` to an open database.

→ *`sql`*
A sort index identifying both the table containing the rows to select from and the manner in which the cursor's rows should be sorted. See "The SELECT Statement" on page 37 for the format of this parameter.

→ *`flags`*
Zero or more flags (OR'd together) that specify how the cursor is to be opened. See "Cursor Open Flags" on page 302 for the set of flags defined for this operation.

→ *`numCategories`*
Number of categories in the *`categoryIDs`* array.

→ *`categoryIDs`*
Array of category IDs used to filter the cursor. If no categories are specified (that is, if *`numCategories`* is 0), no filtering based upon categories is done.

→ *`matchMode`*
One of the following values, indicating how the categories in the *`categoryIDs`* array are applied to the cursor:

`DbMatchAny`
(OR): Include rows with categories matching any of the specified categories.

`DbMatchAll`
(AND): Include rows with categories matching all of the specified categories, including rows with additional category membership.

`DbMatchExact`
Include rows with categories matching exactly the specified categories.

← *`cursorID`*
The ID of the newly-opened cursor. If there was an error opening the cursor, *`*cursorID`* is set to `dbInvalidCursorID`.

Returns Returns `errNone` if the cursor was successfully opened, or one of the following otherwise:

`dmErrInvalidParam`
`dbRef` doesn't reference an open database, *`sql`* is `NULL`, or *`cursorID`* is `NULL`.

`dmErrInvalidCategory`
One or more of the specified category IDs is invalid.

`dmErrInvalidSortIndex`
One of the sort IDs specified in the supplied SQL isn't valid for the specified database table.

`dmErrMemError`
: The operation couldn't be completed due to insufficient memory.

`dmErrNotSchemaDatabase`
: The database specified by *`dbRef`* isn't a schema database.

`dmErrSQLParseError`
: The SQL specified in the *`sql`* parameter is invalid.

`dmErrCursorEOF`
: The cursor was successfully created but the table contains no rows that match the specified criteria.

Comments The *`sql`*, *`flags`*, *`categoryIDs`*, and *`matchMode`* parameters allow your application to specify a subset of the database rows that belong to the cursor. Only the rows that match the specified SQL, flags, and categories (the match mode determines how category matches are applied) will exist in the cursor; those rows are sorted as specified by the sort index.

See Also `DbCursorClose()`, `DbCursorOpen()`

DbCursorRemoveAllRows Function

Purpose Remove all of the cursor's rows from the database.

Declared In `SchemaDatabases.h`

Prototype

```
status_t DbCursorRemoveAllRows
   (uint32_t cursorID)
```

Parameters → *`cursorID`*
: ID of a valid cursor, as returned from `DbCursorOpen()` or `DbCursorOpenWithCategory()`.

Returns Returns `errNone` if the operation completed successfully, or one of the following otherwise:

`dmErrInvalidParam`
: The specified cursor ID isn't valid.

`dmErrRecordBusy`
: One of the rows is in use and cannot be updated.

`dmErrReadOnly`
You've attempted to write to or modify a database that is open in read-only mode.

Comments For each row in the cursor, this function deletes the row's chunk from the database and removes the row entry from the database header.

See Also `DbCursorArchiveAllRows()`, `DbCursorDeleteAllRows()`, `DbRemoveRow()`

DbCursorRequery Function

Purpose Refresh a cursor to reflect any changes made to the database since the last query. If the cursor's contents change, the cursor is repositioned at the first row.

Declared In `SchemaDatabases.h`

Prototype `status_t DbCursorRequery (uint32_t cursorID)`

Parameters → *`cursorID`*
ID of a valid cursor, as returned from `DbCursorOpen()` or `DbCursorOpenWithCategory()`.

Returns Returns `errNone` if the cursor was successfully refreshed, or one of the following otherwise:

`dmErrInvalidParam`
`cursorID` isn't a valid cursor ID or doesn't reference an open cursor.

`dmErrInvalidSortIndex`
The sort index is no longer valid.

`dmErrMemError`
A memory error occurred.

`dmErrCursorEOF`
The cursor contains no rows.

`dmErrIndexOutOfRange`
One or more bindings are no longer valid.

Comments When the cursor is created a snapshot of the row IDs is taken that is used when iterating the cursor's rows. This snapshot of the IDs is not affected by sorting updates due to row modifications or the addition of new rows. `DbCursorRequery()` refreshes the snapshot to reflect any new row additions or sorting changes.

Note that when a refresh occurs the current row may move to a new position (the first row, if the cursor contents change), and future move operations will move from the new position instead of the old position.

See Also `DbCursorOpen()`, `DbCursorUpdate()`

DbCursorSetAbsolutePosition Macro

Purpose Moves the cursor onto the row with the specified index.

Declared In `SchemaDatabases.h`

Prototype `#define DbCursorSetAbsolutePosition (`*`i`*`, `*`o`*`)`

Parameters

→ *i*
: ID of a valid cursor, as returned from `DbCursorOpen()` or `DbCursorOpenWithCategory()`.

→ *o*
: Index of the row to which the cursor should be positioned.

Returns Returns `errNone` if the current row position was moved to a valid row within the cursor, or one of the following otherwise:

`dmErrInvalidParam`
: The specified cursor ID is invalid.

`dmErrCursorBOF`
: An attempt was made to move to a position before the first row in the cursor.

`dmErrCursorEOF`
: An attempt was made to move to a position after the last row in the cursor.

Comments The first row within a cursor has an index value of 1.

Attempting to move before the first row in the cursor or attempting to move past the last row in the cursor generates an error, and the cursor's BOF or EOF property, as appropriate, is set.

When moving through the cursor, rows that have been modified are not moved to their new sort position until `DbCursorRequery()` is called. Similarly any new rows are not available to the cursor until `DbCursorRequery()` is called.

Upon successful completion of the move, any bound variables are updated with corresponding field values for the new current row.

See Also `DbCursorMove()`, `DbCursorMoveToRowID()`

DbCursorUpdate Function

Purpose Write the values in the bound variables to the row at the cursor's current position.

Declared In `SchemaDatabases.h`

Prototype `status_t DbCursorUpdate (uint32_t` *`cursorID`*`)`

Parameters → *`cursorID`*

ID of a valid cursor, as returned from `DbCursorOpen()` or `DbCursorOpenWithCategory()`.

Returns Returns `errNone` if the current row position was successfully moved to the specified row within the cursor, or one of the following otherwise:

`dmErrInvalidParam`
`cursorID` isn't a valid cursor ID or doesn't reference an open cursor.

`dmErrCursorBOF`
The cursor's current position is at BOF, which is not a valid row.

`dmErrCursorEOF`
The cursor's current position is at EOF, which is not a valid row.

`dmErrRecordDeleted`
The current row is marked as deleted.

`dmErrRecordBusy`
The current row is in use and cannot be updated.

`dmErrMemError`
A memory error occurred.

`dmErrWriteOutOfBounds`
The write operation exceeded the bounds of the row.

`dmErrOperationAborted`
The write could not be performed.

Comments Prior to calling `DbCursorUpdate()`, set the bound variables to the desired values. All values are written to the database for the current row. Note that for varying length types (`VarChar` and `Blob`), you should also set the corresponding data size variable (specified when the cursor column was bound to a variable) to indicate the size of the data to be written back to that field.

See Also `DbCursorBindData()`, `DbCursorBindDataWithOffset()`, `DbCursorRequery()`

DbDeleteRow Function

Purpose Delete a row's chunk from a database but leave the row entry in the header and mark the row as deleted for the next HotSync operation.

Declared In `SchemaDatabases.h`

Prototype
```
status_t DbDeleteRow (DmOpenRef dbRef,
   uint32_t rowID)
```

Parameters
→ *`dbRef`*
`DmOpenRef` to an open database.

→ *`rowID`*
Row ID or cursor ID identifying the row to be deleted.

Returns Returns `errNone` if the operation completed successfully, or one of the following otherwise:

`dmErrInvalidParam`
`dbRef` doesn't reference an open database, or *`rowID`* isn't a valid cursor or row ID.

`dmErrNotSchemaDatabase`
`dbRef` doesn't reference a schema database.

`dmErrReadOnly`
The specified database is opened in read-only mode.

`dmErrUniqueIDNotFound`
`rowID` doesn't identify a valid row within the database.

`dmErrIndexOutOfRange`
`rowID` doesn't identify a valid row within the database.

`dmErrRecordDeleted`
: The specified record is already marked as deleted.

`dmErrRecordArchived`
: The specified record is marked as archived.

`dmErrRecordBusy`
: The specified record is in use.

`dmErrCorruptDatabase`
: The database is corrupt.

Comments This function deletes the row's chunk from the database but leaves the row entry in the header and marks the row as deleted. During the next HotSync operation, a conduit can save the row data on the desktop and then remove those row entries in the header that are marked as deleted.

See Also `DbArchiveRow()`, `DbCursorDeleteAllRows()`, `DbInsertRow()`, `DbRemoveRow()`

DbEnableSorting Function

Purpose Turn automatic sorting on or off for a given database.

Declared In `SchemaDatabases.h`

Prototype
```
status_t DbEnableSorting (DmOpenRef dbRef,
   Boolean enable)
```

Parameters

→ *`dbRef`*
: `DmOpenRef` to an open database.

→ *`enable`*
: If `true`, sorting is enabled. If `false`, sorting is disabled.

Returns Returns `errNone` if the operation completed successfully, or one of the following if an error occurred:

`dmErrInvalidParam`
: *`dbRef`* doesn't reference an open database.

`dmErrNotSchemaDatabase`
: The specified database is not a schema database.

`dmErrReadOnly`
: The database is not open for write access.

`dmErrInvalidOperation`
The specified database has no sort indices defined.

Comments If *`enable`* is to `true` and automatic sorting was previously turned off, the database is resorted, making all current row indices invalid.

If you don't have authorization to modify the database, this function does nothing.

This function sorts the database according to each defined sort index.

See Also `DbAddSortIndex()`, `DbIsSortingEnabled()`

DbGetAllColumnDefinitions Function

Purpose Retrieve all of a table's column definitions.

Declared In `SchemaDatabases.h`

Prototype

```
status_t DbGetAllColumnDefinitions
   (DmOpenRef dbRef, const char *table,
   uint32_t *numColumnsP,
   DbSchemaColumnDefnType **columnDefnsPP)
```

Parameters

→ *`dbRef`*
`DmOpenRef` to an open database.

→ *`table`*
Table name.

← *`numColumnsP`*
Number of elements in the *`*columnDefnsPP`* array.

← *`columnDefnsPP`*
Pointer to an array of `DbSchemaColumnDefnType` structures, each representing a single column definition. The Data Manager allocates the array and returns a pointer to it.

Returns Returns `errNone` if the operation completed successfully, or one of the following if there was an error:

`dmErrInvalidParam`
`dbRef` doesn't reference an open database, *`numColumnsP`* is `NULL`, *`columnDefnsPP`* is `NULL`, or *`table`* is `NULL`.

`dmErrNotSchemaDatabase`
The specified database is not a schema database.

`dmErrMemError`
: The function was unable to allocate sufficient memory to contain the column definitions.

`dmErrInvalidTableName`
: The database doesn't contain a table with the specified name.

dmErrNoData
: The specified table has no columns defined.

`dmErrOneOrMoreFailed`
: At least one of the column definitions could not be retrieved.

Comments Your application is responsible for releasing the array allocated by this call. To do this, use `DbReleaseStorage()`. After `DbReleaseStorage()` is called, the references returned by `DbGetAllColumnDefinitions()` must be considered invalid since the underlying storage may have been relocated.

See Also `DbAddColumn()`, `DbGetColumnDefinitions()`

DbGetAllColumnPropertyValues Function

Purpose Retrieve all of a table's column property values.

Declared In `SchemaDatabases.h`

Prototype

```
status_t DbGetAllColumnPropertyValues
    (DmOpenRef dbRef, const char *table,
    Boolean customPropsOnly, uint32_t *numPropsP,
    DbColumnPropertyValueType **propValuesPP)
```

Parameters

→ *`dbRef`*
: `DmOpenRef` to an open database.

→ *`table`*
: Table name.

→ *`customPropsOnly`*
: If `true`, only user-defined custom column property values are retrieved. Otherwise, all default (built-in) and custom column property values are retrieved.

← *`numPropsP`*
: Number of elements in the *`*propValuesPP`* array.

← *`propValuesPP`*
Pointer to an array of `DbColumnPropertyValueType` structures, each representing a single column property value. The Data Manager allocates the array and returns a pointer to it.

Returns Returns `errNone` if the property value was successfully retrieved, or one of the following if an error occurred:

`dmErrInvalidParam`
`dbRef` doesn't reference an open database, *`numPropsP`* is `NULL`, *`propValuesPP`* is `NULL`, or *`table`* is `NULL`.

`dmErrNotSchemaDatabase`
The specified database is not a schema database.

`dmErrInvalidTableName`
The database doesn't contain a table with the specified name.

`dmErrMemError`
A memory error occurred.

`dmErrInvalidColumnID`
The specified table has no defined columns.

`memErrNotEnoughSpace`
A memory error occurred.

Comments The *`customPropsOnly`* argument controls whether all properties or just custom properties are retrieved. Default properties include: `dbColumnNameProperty`, `dbColumnDatatypeProperty`, `dbColumnSizeProperty` and `dbColumnAttribProperty`.

Your application is responsible for releasing the array allocated by this call. To do this, use `DbReleaseStorage()`. After `DbReleaseStorage()` is called, the references returned by `DbGetAllColumnPropertyValues()` must be considered invalid since the underlying storage may have been relocated.

See Also `DbGetColumnPropertyValue()`, `DbGetColumnPropertyValues()`, `DbSetColumnPropertyValues()`

DbGetAllColumnValues Function

Purpose Retrieve all column values for a specified row.

Declared In `SchemaDatabases.h`

Prototype
```
status_t DbGetAllColumnValues (DmOpenRef dbRef,
    uint32_t rowID, uint32_t *numColumnsP,
    DbSchemaColumnValueType **columnValuesPP)
```

Parameters

→ *`dbRef`*
`DmOpenRef` to an open database.

→ *`rowID`*
Row ID or cursor ID identifying the row for which column values are to be retrieved.

← *`numColumnsP`*
The number of retrieved column values.

← *`columnValuesPP`*
Pointer to an array of structures, each representing a single column value. The Data Manager allocates the array and returns a pointer to it.

Returns Returns `errNone` if successful, or one of the following if an error occurred:

`dmErrInvalidParam`
`rowID` is not a row or cursor ID, *`dbRef`* doesn't reference an open database, or *`columnValuesPP`* is `NULL`.

`dmErrCursorBOF`
The supplied cursor ID is BOF.

`dmErrCursorEOF`
The supplied cursor ID is EOF.

`dmErrUniqueIDNotFound`
The supplied cursor ID represents an invalid row.

`dmErrNotSchemaDatabase`
The specified database is not a schema database.

`dmErrUniqueIDNotFound`
The supplied row or cursor ID doesn't correspond to a row within the database.

`dmErrRecordDeleted`
The specified row is marked as deleted.

`dmErrInvalidColSpec`
: There are no columns defined for the specified table.

`dmErrNoColumnData`
: The specified row has no data.

`dmErrMemError`
: A memory error occurred.

Comments Your application is responsible for releasing the array allocated by this call. To do this, use `DbReleaseStorage()`. After `DbReleaseStorage()` is called, the references returned by `DbGetAllColumnValues()` must be considered invalid since the underlying storage may have been relocated.

See Also `DbCopyColumnValues()`, `DbGetColumnValue()`, `DbGetColumnValues()`, `DbWriteColumnValues()`

DbGetCategory Function

Purpose Retrieve the category membership for the specified row.

Declared In `SchemaDatabases.h`

Prototype

```
status_t DbGetCategory (DmOpenRef dbRef,
   uint32_t rowID, uint32_t *pNumCategories,
   CategoryID *pCategoryIDs[])
```

Parameters

→ *`dbRef`*
: `DmOpenRef` to an open database.

→ *`rowID`*
: Row ID or cursor ID identifying the row for which to get categories.

← *`pNumCategories`*
: The number of elements in the *`pCategoryIDs`* array.

← *`pCategoryIDs`*
: Array of category IDs. The specified row is a member of each of the categories in this list. Pass `NULL` for this parameter if all you want is the number of categories of which this row is a member.

Returns Returns `errNone` if no error, or one of the following if an error occurs:

`dmErrInvalidParam`
: *`dbRef`* doesn't reference an open database, or the specified row or cursor ID is not valid.

`dmErrNotSchemaDatabase`
: The specified database is not a schema database.

`dmErrIndexOutOfRange`
: The specified row or cursor ID doesn't reference a row within the table.

`dmErrRecordDeleted`
: The specified row is marked as deleted.

`dmErrMemError`
: A memory error occurred.

Comments Your application is responsible for releasing the array allocated by this call. To do this, use `DbReleaseStorage()`. After `DbReleaseStorage()` is called, the references returned by `DbGetCategory()` must be considered invalid since the underlying storage may have been relocated.

If the specified row isn't a member of any categories, this function sets **`pNumCategories`* to 0 and *`*pCategoryIDs`* to `NULL`.

See Also `DbAddCategory()`, `DbIsRowInCategory()`, `DbSetCategory()`

DbGetColumnDefinitions Function

Purpose Retrieve one or more table column definitions.

Declared In `SchemaDatabases.h`

Prototype

```
status_t DbGetColumnDefinitions (DmOpenRef dbRef,
   const char *table, uint32_t numColumns,
   const uint32_t columnIDs[],
   DbSchemaColumnDefnType **columnDefnsPP)
```

Parameters

→ *`dbRef`*
: `DmOpenRef` to an open database.

→ *`table`*
: Table name.

→ *`numColumns`*
: The number of columns in the *`columnIDs`* array.

→ *`columnIDs`*
: Array of column IDs, indicating the columns for which definitions are to be retrieved.

← *`columnDefnsPP`*
: Pointer to an array of `DbSchemaColumnDefnType` structures; each array element contains the definition for a column.

Returns Returns `errNone` if the operation completed successfully, or one of the following if there was an error:

`dmErrInvalidParam`
: *`dbRef`* doesn't reference an open database, *`columnIDs`* is `NULL`, *`columnDefnsPP`* is `NULL`, or *`table`* is `NULL`.

`dmErrNotSchemaDatabase`
: The specified database is not a schema database.

`dmErrMemError`
: The function was unable to allocate sufficient memory to contain the column definitions.

`dmErrInvalidTableName`
: The database doesn't contain a table with the specified name.

`dmErrInvalidColumnID`
: The specified table has no columns defined.

`dmErrOneOrMoreFailed`
: At least one of the column definitions could not be retrieved.

Comments Your application is responsible for releasing the array allocated by this call. To do this, use `DbReleaseStorage()`. After `DbReleaseStorage()` is called, the references returned by `DbGetColumnDefinitions()` must be considered invalid since the underlying storage may have been relocated.

See Also `DbAddColumn()`, `DbGetAllColumnDefinitions()`

DbGetColumnID Function

Purpose Retrieve the column ID for a column index.

Declared In `SchemaDatabases.h`

Prototype
```
status_t DbGetColumnID (DmOpenRef dbRef,
   const char *table, uint32_t columnIndex,
   uint32_t *columnIDP)
```

Parameters

→ *dbRef*
DmOpenRef to an open database.

→ *table*
Table name.

→ *columnIndex*
The index of the column for which the ID is being retrieved.

← *columnIDP*
The column ID.

Returns Returns `errNone` if the column ID was successfully retrieved, or one of the following if an error occurred:

`dmErrInvalidParam`
dbRef doesn't reference an open database, or *table* is NULL.

`dmErrNotSchemaDatabase`
The specified database isn't a schema database.

`dmErrInvalidTableName`
The database doesn't contain a table with the specified name.

`dmErrColumnIndexOutOfRange`
The supplied column index exceeds the number of columns in the table.

Comments See the Comments section under `DbNumColumns()` for an example of how you use this function.

See Also `DbNumColumns()`

DbGetColumnPropertyValue Function

Purpose Retrieve the value of a specified table column property.

Declared In `SchemaDatabases.h`

Prototype
```
status_t DbGetColumnPropertyValue
    (DmOpenRef dbRef, const char *table,
    uint32_t columnID,
    DbSchemaColumnProperty propID,
    uint32_t *numBytesP, void **propValuePP)
```

Parameters

→ *`dbRef`*
`DmOpenRef` to an open database.

→ *`table`*
Table name.

→ *`columnID`*
The ID of the column for which the property is being retrieved.

→ *`propID`*
The ID of the property being retrieved.

← *`numBytesP`*
The size, in bytes, of the retrieved property value.

← *`propValuePP`*
The retrieved property value.

Returns Returns `errNone` if the property value was successfully retrieved, or one of the following if an error occurred:

`dmErrInvalidParam`
`dbRef` doesn't reference an open database, *`numBytesP`* is `NULL`, *`propValuePP`* is `NULL`, or *`table`* is `NULL`.

`dmErrNotSchemaDatabase`
The specified database is not a schema database.

`dmErrInvalidTableName`
The database doesn't contain a table with the specified name.

`dmErrInvalidColumnID`
The specified table has no defined columns, or the specified column index is not a defined column.

`dmErrInvalidPropID`
: The column doesn't have a property with the specified property ID.

`memErrNotEnoughSpace`
: A memory error occurred.

Comments Your application is responsible for releasing the memory allocated by this call to contain the property value. To do this, use `DbReleaseStorage()`. After `DbReleaseStorage()` is called, the references returned by `DbGetColumnPropertyValue()` must be considered invalid since the underlying storage may have been relocated.

See Also `DbGetAllColumnPropertyValues()`, `DbGetColumnPropertyValues()`, `DbSetColumnPropertyValue()`

DbGetColumnPropertyValues Function

Purpose Retrieve the value of one or more table column properties.

Declared In `SchemaDatabases.h`

Prototype

```
status_t DbGetColumnPropertyValues
    (DmOpenRef dbRef, const char *table,
    uint32_t numProps,
    const DbColumnPropertySpecType propSpecs[],
    DbColumnPropertyValueType **propValuesPP)
```

Parameters

→ *`dbRef`*
: `DmOpenRef` to an open database.

→ *`table`*
: Table name.

→ *`numProps`*
: The number of elements in the *`propSpecs`* array.

→ *`propSpecs`*
: Array of column ID/property ID pairs. See "DbColumnPropertySpecType" on page 291.

← *`propValuesPP`*
: Array of property values. See "DbColumnPropertyValueType" on page 292.

Returns Returns `errNone` if the property value was successfully retrieved, or one of the following if an error occurred:

`dmErrInvalidParam`
: *`dbRef`* doesn't reference an open database, *`numProps`* is zero, *`propSpecs`* is `NULL`, *`propValuePP`* is `NULL`, or *`table`* is `NULL`.

`dmErrNotSchemaDatabase`
: The specified database is not a schema database.

`dmErrInvalidTableName`
: The database doesn't contain a table with the specified name.

`dmErrMemError`
: A memory error occurred.

`dmErrInvalidColumnID`
: The specified table has no defined columns, or the at least one of the specified column indices is not a defined column.

`dmErrInvalidPropID`
: At least one column doesn't have a property with the specified property ID.

`memErrNotEnoughSpace`
: A memory error occurred.

Comments Your application is responsible for releasing the array allocated by this call. To do this, use `DbReleaseStorage()`. After `DbReleaseStorage()` is called, the references returned by `DbGetColumnPropertyValues()` must be considered invalid since the underlying storage may have been relocated.

See Also `DbGetAllColumnPropertyValues()`, `DbGetColumnPropertyValue()`, `DbSetColumnPropertyValues()`

DbGetColumnValue Function

Purpose Retrieve a single column value for a row.

Declared In `SchemaDatabases.h`

Prototype

```
status_t DbGetColumnValue (DmOpenRef dbRef,
   uint32_t rowID, uint32_t columnID,
   uint32_t offset, void **valuePP,
   uint32_t *valueSizeP)
```

Parameters

→ *dbRef*
: `DmOpenRef` to an open database.

→ *rowID*
: Row ID or cursor ID identifying the row for which the column value is to be retrieved.

→ *columnID*
: The column ID.

→ *offset*
: For variable-length columns, the column value offset from which data is retrieved. This value is interpreted as a byte offset.

← *valuePP*
: The column value.

← *valueSizeP*
: The size of the column value, in bytes.

Returns Returns `errNone` if successful, or one of the following if an error occurred:

`dmErrInvalidParam`
: *rowID* is not a row or cursor ID, or *valuePP* is `NULL`.

`dmErrCursorBOF`
: The supplied cursor ID is BOF.

`dmErrCursorEOF`
: The supplied cursor ID is EOF.

`dmErrUniqueIDNotFound`
: The supplied cursor ID represents an invalid row.

`dmErrNotSchemaDatabase`
: The specified database is not a schema database.

`dmErrUniqueIDNotFound`
: The supplied row or cursor ID doesn't correspond to a row within the database.

`dmErrRecordDeleted`
: The specified row is marked as deleted.

`dmErrInvalidColSpec`
: There are no columns defined for the specified table.

`dmErrInvalidColumnID`
: The supplied column ID is invalid.

`dmErrNoColumnData`
: The specified row has no data for the column.

`dmErrReadOutOfBounds`
: The specified offset exceeds the bounds of the column.

`dmErrBufferNotLargeEnough`
: The supplied buffer isn't large enough to contain the column value.

`dmErrMemError`
: A memory error occurred.

Comments This function returns a reference to the column data. Offset-based reads are not supported for fixed-length column data types; the offset parameter is ignored for these data types. The column data types that support offset-based reads are:

- `VarChar`
- `Blob`
- `Vector`

Your application is responsible for releasing the column value buffer allocated by this call. To do this, use `DbReleaseStorage()`.

See Also `DbCopyColumnValue()`, `DbGetAllColumnValues()`, `DbGetColumnValues()`, `DbWriteColumnValue()`

DbGetColumnValues Function

Purpose Retrieve one or more column values for a row.

Declared In `SchemaDatabases.h`

Prototype

```
status_t DbGetColumnValues (DmOpenRef dbRef,
   uint32_t rowID, uint32_t numColumns,
   const uint32_t columnIDs,
   DbSchemaColumnValueType **columnValuesPP)
```

Parameters

→ *`dbRef`*
`DmOpenRef` to an open database.

→ *`rowID`*
Row ID or cursor ID identifying the row for which the column values are to be retrieved.

→ *`numColumns`*
The number of elements in the *`columnIDs`* array.

→ *`columnIDs`*
Array of one or more column IDs indicating the columns for which values are to be retrieved.

← *`columnValuesPP`*
An array of data structures containing the retrieved column values.

Returns Returns `errNone` if successful, or one of the following if an error occurred:

`dmErrInvalidParam`
`rowID` is not a row or cursor ID, *`dbRef`* doesn't reference an open database, *`numColumns`* is zero, or *`columnValuesPP`* is `NULL`.

`dmErrCursorBOF`
The supplied cursor ID is BOF.

`dmErrCursorEOF`
The supplied cursor ID is EOF.

`dmErrUniqueIDNotFound`
The supplied cursor ID represents an invalid row.

`dmErrNotSchemaDatabase`
The specified database is not a schema database.

`dmErrUniqueIDNotFound`
: The supplied row or cursor ID doesn't correspond to a row within the database.

`dmErrRecordDeleted`
: The specified row is marked as deleted.

`dmErrInvalidColSpec`
: There are no columns defined for the specified table.

`dmErrInvalidColumnID`
: The one or more of the specified column IDs is invalid.

`dmErrNoColumnData`
: The specified row has no data.

`dmErrBufferNotLargeEnough`
: At least one of the supplied buffers isn't large enough to contain the corresponding column value.

`dmErrMemError`
: A memory error occurred.

Comments Your application is responsible for releasing the array allocated by this call. To do this, use `DbReleaseStorage()`. After `DbReleaseStorage()` is called, the references returned by `DbGetColumnValues()` must be considered invalid since the underlying storage may have been relocated.

See Also `DbCopyColumnValues()`, `DbGetAllColumnValues()`, `DbGetColumnValue()`, `DbWriteColumnValues()`

DbGetRowAttr Function

Purpose Retrieve a row's attributes.

Declared In `SchemaDatabases.h`

Prototype

```
status_t DbGetRowAttr (DmOpenRef dbRef,
   uint32_t rowID, uint16_t *attrP)
```

Parameters

→ *`dbRef`*
: `DmOpenRef` to an open database.

→ *`rowID`*
: Row ID or cursor ID identifying the row for which attributes are to be retrieved.

← *attrP*
The row's attributes. See "Schema Database Row Attributes" on page 300 for the set of attributes that can be retrieved.

Returns Returns `errNone` if the row's attributes were successfully retrieved, or one of the following if an error occurred:

`dmErrNotRecordDB`
You've attempted to perform a row function on a resource database.

`dmErrIndexOutOfRange`
The specified index is out of range.

See Also `DbGetTableForRow()`, `DbSetRowAttr()`

DbGetRuleSet Function

Purpose Get the current access rules for a secure database.

Declared In `SchemaDatabases.h`

Prototype
```
status_t DbGetRuleSet (DatabaseID dbID,
   AzmRuleSetType *ruleset)
```

Parameters
→ *dbID*
ID of the secure database for which access rules are to be retrieved.

← *ruleset*
Handle to the database's access rules.

Returns Returns `errNone` if the operation completed successfully, or one of the following if an error occurred:

`dmErrInvalidParam`
dbID doesn't reference a database or *ruleset* is `NULL`.

`dmErrNotSecureDatabase`
The specified database is not a secure database.

`dmErrAccessDenied`
You don't have sufficient privileges to obtain the database's access rules.

Comments The database must exist, and must be a secure database.

This function requires that the calling application to be authorized for the Modify action as defined by the Authorization Manager (that is, it must be the application that created the secure database). If the application does not have modification rights, the function fails with `dmErrAccessDenied`.

See Also `DbCreateSecureDatabase()`, `DbCreateSecureDatabaseFromImage()`

DbGetSortDefinition Function

Purpose Get a sort index given its position in the list of sort indices defined for a database.

Declared In `SchemaDatabases.h`

Prototype

```
status_t DbGetSortDefinition (DmOpenRef dbRef,
   uint32_t sortIndex, char **orderByPP)
```

Parameters

→ *`dbRef`*
`DmOpenRef` to an open database.

→ *`sortIndex`*
An integer index value, ranging from 0 to one less than the value returned from `DbNumSortIndexes()`, indicating which sort index is desired.

← *`orderByPP`*
Upon return, `*`*`orderByPP`* points to the SQL string that makes up the sort index.

Returns Returns `errNone` if the operation succeeded, or one of the following otherwise:

`dmErrInvalidParam`
`dbRef` doesn't reference an open database.

`dmErrNotSchemaDatabase`
The specified database is not a schema database.

`dmErrInvalidIndex`
The *`sortIndex`* parameter is greater than the highest sort index value defined for this database.

Comments See the Comments section under `DbNumSortIndexes()` for an example of how you use this function.

DbGetTableForRow Function

Purpose Obtain the name of the table that contains a specified row.

Declared In `SchemaDatabases.h`

Prototype

```
status_t DbGetTableForRow (DmOpenRef dbRef,
   uint32_t rowID, char *buf, size_t bufSize)
```

Parameters

→ *`dbRef`*
: `DmOpenRef` to an open database.

→ *`rowID`*
: Row ID or cursor ID identifying the row for which the table is to be determined.

← *`buf`*
: Pass a pointer to the buffer into which the table name is to be written.

→ *`bufSize`*
: The size of *`buf`*, in bytes.

Returns Returns `errNone` if the operation succeeded, or one of the following otherwise:

`dmErrInvalidParam`
: *`dbRef`* doesn't reference an open database, *`rowID`* isn't a row or cursor ID, *`buf`* is `NULL`, or *`bufSize`* is zero.

`dmErrNotSchemaDatabase`
: The specified database is not a schema database.

`dmErrUniqueIDNotFound`
: The specified row or cursor ID doesn't correspond to a row in the database.

`dmErrMemError`
: The supplied buffer isn't large enough to contain the table name, or another memory error occurred.

See Also `DbCursorGetCurrentRowID()`

DbGetTableName Function

Purpose Obtain a table’s name, given the index of the table within a database.

Declared In `SchemaDatabases.h`

Prototype
```
status_t DbGetTableName (DmOpenRef dbRef,
   uint32_t index, char *table)
```

Parameters

→ *`dbRef`*
: `DmOpenRef` to an open database.

→ *`index`*
: Index of the table within the database.

← *`table`*
: Table name.

Returns Returns `errNone` if the operation succeeded, or one of the following otherwise:

`dmErrInvalidParam`
: *`dbRef`* doesn’t reference an open database, or *`table`* is `NULL`.

`dmErrNotSchemaDatabase`
: The specified database is not a schema database.

`dmErrSchemaIndexOutOfRange`
: The specified index is greater than the number of tables in the database.

Comments Table indices are zero-based. That is, the first table in a database has an index value of zero.

See Also `DbNumTables()`

DbGetTableSchema Function

Purpose Get the schema for a table, including the definitions and properties for all of the table’s columns.

Declared In `SchemaDatabases.h`

Prototype
```
status_t DbGetTableSchema (DmOpenRef dbRef,
    const char *table,
    DbTableDefinitionType **schemaPP)
```

Parameters

→ *`dbRef`*
: `DmOpenRef` to an open database.

→ *`table`*
: Table name.

← *`schemaPP`*
: The schema. Allocate a pointer to a `DbTableDefinitionType` structure and supply the address of this pointer when calling `DbGetTableSchema()`. Upon return, your pointer variable contains the address of a `DbTableDefinitionType` structure containing the table name, the number of columns in the table, and a pointer to the first element in an array of column definition.

Returns Returns `errNone` if the schema was successfully retrieved, or one of the following if an error occurred:

`dmErrInvalidParam`
: *`dbRef`* doesn’t reference an open databases, no table name was specified, or *`schemaPP`* is `NULL`.

`dmErrNotSchemaDatabase`
: The specified database is not a schema database.

`dmErrInvalidTableName`
: The database doesn’t contain a table with the specified name.

`dmErrMemError`
: A memory error occurred.

Comments Your application is responsible for releasing the buffer pointed to by **`schemaPP`*. To do this, use `DbReleaseStorage()`. After `DbReleaseStorage()` is called, the references returned by `DbGetTableSchema()` must be considered invalid since the underlying storage may have been relocated.

See Also `DbGetTableName()`, `DbHasTable()`

DbHasSortIndex Function

Purpose Determine whether a particular sort index has been defined for a database.

Declared In `SchemaDatabases.h`

Prototype
```
Boolean DbHasSortIndex (DmOpenRef dbRef,
    const char *orderBy)
```

Parameters

→ *`dbRef`*
`DmOpenRef` to an open database.

→ *`orderBy`*
The sort index being checked for. See "The SELECT Statement" on page 37 for the format of this parameter.

Returns Returns `errNone` if the operation completed successfully, or one of the following if an error occurred:

`dmErrInvalidParam`
`dbRef` doesn't reference an open database.

`dmErrNotSchemaDatabase`
The specified database is not a schema database.

`dmErrSQLParseError`
The specified table name or the sort information specified in the sort index is invalid.

See Also `DbAddSortIndex()`, `DbRemoveSortIndex()`

DbHasTable Function

Purpose Determine whether a specific table exists in a particular database.

Declared In `SchemaDatabases.h`

Prototype
```
Boolean DbHasTable (DmOpenRef dbRef,
    const char *table)
```

Parameters

→ *`dbRef`*
`DmOpenRef` to an open database.

→ *`table`*
Table name.

Returns Returns `true` if the specified database contains the named table. Returns `false` if either the table doesn't exist in the database, *dbRef* is not a valid reference to an open database, or the specified database is not a schema database.

See Also DbGetTableName(), DbGetTableSchema()

DbInsertRow Function

Purpose Add a row to a specified database table.

Declared In `SchemaDatabases.h`

Prototype

```
status_t DbInsertRow (DmOpenRef dbRef,
   const char *table, uint32_t numColumnValues,
   DbSchemaColumnValueType *columnValuesP,
   uint32_t *rowIDP)
```

Parameters

→ *dbRef*
  `DmOpenRef` to an open database.

→ *table*
  Table name.

→ *numColumnValues*
  Number of column values in the *columnValuesP* array.

→ *columnValuesP*
  Array of column values, where each value represents a column value for the new row.

← *rowIDP*
  Row ID of the newly added row, or `dbInvalidRowID` if the row couldn’t be added.

Returns Returns `errNone` if the row was added successfully, or one of the following otherwise:

`dmErrInvalidParam`
  dbRef doesn’t reference an open database.

`dmErrInvalidTableName`
  The specified table name is invalid.

`dmErrNotSchemaDatabase`
  The specified database is not a schema database.

`dmErrReadOnly`
  The database is not open for write access.

`dmErrInvalidColSpec`
: One or more column values doesn't fit in the corresponding column.

`dmErrInvalidColumnID`
: The number of column values supplied exceeds the number of columns in the table.

`dmErrMemError`
: A memory error occurred.

Comments The new row is added to the end of the database. Any open cursors are not updated; use `DbCursorRequery()` to update a particular cursor's contents.

If *`numColumnValues`* is zero or *`columnValuesP`* is `NULL`, an empty row is created which may subsequently be written into using either `DbWriteColumnValue()` or `DbWriteColumnValues()`.

See Also `DbArchiveRow()`, `DbDeleteRow()`, `DbRemoveRow()`

DbIsCursorID Function

Purpose Determine whether a specified ID is a cursor ID.

Declared In `SchemaDatabases.h`

Prototype `Boolean DbIsCursorID (uint32_t` *`uniqueID`*`)`

Parameters → *`uniqueID`*
: The ID to be checked.

Returns Returns `true` if *`uniqueID`* is a cursor ID, `false` otherwise.

Comments Cursor IDs can generally be used interchangeably with row IDs. If you are using a cursor, however, it is more efficient to use a cursor ID.

See Also `DbIsRowID()`

DbIsRowID Function

Purpose Determine whether a specified ID is a row ID.

Declared In `SchemaDatabases.h`

Prototype `Boolean DbIsRowID (uint32_t` *`uniqueID`*`)`

Parameters → *`uniqueID`*
The ID to be checked.

Returns Returns `true` if *`uniqueID`* is a row ID, `false` otherwise.

Comments Cursor IDs can generally be used interchangeably with row IDs. If you are using a cursor, however, it is more efficient to use a cursor ID.

See Also `DbIsCursorID()`

DbIsRowInCategory Function

Purpose Determine whether a row is a member of the specified categories, depending on the given match mode criteria.

Declared In `SchemaDatabases.h`

Prototype

```
status_t DbIsRowInCategory (DmOpenRef dbRef,
   uint32_t rowID, uint32_t numCategories,
   const CategoryID categoryIDs[],
   DbMatchModeType matchMode,
   Boolean *pIsInCategory)
```

Parameters → *`dbRef`*
`DmOpenRef` to an open database.

→ *`rowID`*
Row ID or cursor ID identifying the row for which category membership is to be checked.

→ *`numCategories`*
Number of categories in the *`categoryIDs`* array.

→ *`categoryIDs`*
Array of category ID values.

→ *`matchMode`*
One of the following values:

`DbMatchAny`
: (OR) Set *`pIsInCategory`* to `true` if the row membership includes any of the categories specified in the *`categoryIDs`* array.

`DbMatchAll`
: (AND) Set *`pIsInCategory`* to `true` if the row membership includes all of the categories specified in the *`categoryIDs`* array, including rows with additional category membership.

`DbMatchExact`
: Set *`pIsInCategory`* to `true` if the row membership exactly matches the categories specified in the *`categoryIDs`* array.

← *`pIsInCategory`*
: `true` if the row at the given index position has membership in the given category set according to the supplied match mode value. `false` otherwise.

Returns Returns `errNone` if the operation completed successfully, or one of the following otherwise:

`dmErrInvalidParam`
: *`dbRef`* doesn't reference an open database, *`rowID`* isn't a row or cursor ID, *`numCategories`* is zero and *`categoryIDs`* is not `NULL`, *`numCategories`* is nonzero and *`categoryIDs`* is `NULL`, or *`matchMode`* isn't one of the allowable values.

`dmErrNotSchemaDatabase`
: The specified database is not a schema database.

`dmErrUniqueIDNotFound`
: The specified row ID doesn't reference a row within the database.

`dmErrRecordDeleted`
: The indicated row is marked as deleted.

`dmErrMemError`
: A memory error occurred.

Comments To check whether a row has no category membership (that is, it belongs to the "Unfiled" category), set *`numCategories`* to 0 and *`categoryIDs`* to `NULL`.

This function might always return `false` if

- none of the supplied category IDs is a valid category ID, and the supplied match mode criteria value is `DbMatchAny`.
- any of the supplied category IDs is *not* a valid category ID, and the supplied match mode criteria value is either `DbMatchAll` or `DbMatchExact`.

See Also `DbGetCategory()`

DbIsSortingEnabled Function

Purpose Determine whether a given database keeps its contents sorted according to one or more sort indices.

Declared In `SchemaDatabases.h`

Prototype
```
status_t DbIsSortingEnabled (DmOpenRef dbP,
   Boolean *enableP)
```

Parameters

→ *`dbP`*
: `DmOpenRef` to an open database.

← *`enableP`*
: `true` if the database contents are kept sorted, `false` otherwise.

Returns Returns `errNone` if the operation completed successfully, or one of the following if an error occurred:

`dmErrInvalidParam`
: *`dbRef`* doesn't reference an open database.

`dmErrNotSchemaDatabase`
: The specified database is not a schema database.

See Also `DbEnableSorting()`

DbMoveCategory Function

Purpose Change the category membership for rows meeting a set of category criteria to a specified category.

Declared In `SchemaDatabases.h`

Prototype
```
status_t DbMoveCategory (DmOpenRef dbRef,
   CategoryID toCategory,
   uint32_t numFromCategories,
   const CategoryID fromCategoryIDs[],
   DbMatchModeType matchMode)
```

Parameters

→ *`dbRef`*
: `DmOpenRef` to an open database.

→ *`toCategory`*
: Category ID to which row membership should be moved.

→ *`numFromCategories`*
: Number of elements in the *`fromCategoryIDs`* array.

→ *`fromCategoryIDs`*
: Array of category ID values from which row membership is to be moved.

→ *`matchMode`*
: One of the following values:

`DbMatchAny`
: (OR) Replace category membership for rows with membership that includes any of the categories specified in the *`fromCategoryIDs`* array.

`DbMatchAll`
: (AND) Replace category membership for rows with membership that includes all of the categories specified in the *`fromCategoryIDs`* array, including rows with additional category membership.

`DbMatchExact`
: Replace category membership for rows with membership that exactly matches the categories specified in the *`fromCategoryIDs`* array.

Returns Returns `errNone` if the operation completed successfully, or one of the following if an error occurred:

`dmErrInvalidParam`
: *`dbRef`* doesn't reference an open database, *`numCategories`* is zero and *`fromCategoryIDs`* is not `NULL`, *`numCategories`* is nonzero and *`fromCategoryIDs`* is `NULL`, or *`matchMode`* isn't one of the allowable values.

`dmErrNotSchemaDatabase`
: The specified database is not a schema database.

`dmErrReadOnly`
: The specified database is a read-only database or is open in read-only mode.

`dmErrMemError`
: A memory error occurred.

`dmErrInvalidCategory`
: One or more of the specified categories is not a valid category.

`dmErrRecordBusy`
: At least one of the database's rows is in use and cannot be updated.

Comments The database must be opened with write access.

An application can also move row membership from no membership ("Unfiled") to membership in a single category by

- specifying a valid category ID value for the *`toCategory`* parameter, AND
- specifying `NULL` for the *`fromCategoryIDs`* parameter and 0 for *`numFromCategories`*. In this case, the *`matchMode`* parameter is ignored.

This function might perform no action if

- none of the category IDs in *`fromCategoryIDs`* are valid and the match mode criteria value is `DbMatchAny`.
- any of the category IDs in *`fromCategoryIDs`* are not valid and the match mode criteria value is either `DbMatchAll` or `DbMatchExact`.

See Also `DbRemoveCategoryAllRows()`

DbNumCategory Function

Purpose Determine how many categories a specified row is a member of.

Declared In `SchemaDatabases.h`

Prototype
```
status_t DbNumCategory (DmOpenRef dbRef,
   uint32_t rowID, uint32_t *pNumCategories)
```

Parameters

→ *`dbRef`*
: `DmOpenRef` to an open database.

→ *`rowID`*
: Row ID or cursor ID identifying the row being analyzed.

← *`pNumCategories`*
: The number of categories of which the row is a member.

Returns Returns `errNone` if the operation completed successfully, or one of the following if an error occurred:

`dmErrInvalidParam`
: *`dbRef`* doesn't reference an open database, *`rowID`* isn't a row or cursor ID, or *`pNumCategories`* is `NULL`.

`dmErrNotSchemaDatabase`
: The specified database is not a schema database.

`dmErrUniqueIDNotFound`
: The specified row ID doesn't reference a row within the database.

`dmErrRecordDeleted`
: The indicated row is marked as deleted.

`dmErrMemError`
: A memory error occurred.

See Also `DbGetCategory()`

DbNumColumns Function

Purpose Get the number of columns in a specified table.

Declared In `SchemaDatabases.h`

Prototype
```
status_t DbNumColumns (DmOpenRef dbRef,
   const char *table, uint32_t *columnCountP)
```

Parameters

→ *dbRef*
: `DmOpenRef` to an open database.

→ *table*
: Table name.

← *columnCountP*
: The number of columns in the table.

Returns Returns `errNone` if the operation completed successfully, or one of the following if there was an error:

`dmErrInvalidParam`
: *dbRef* doesn't reference an open database or *table* is `NULL`.

`dmErrNotSchemaDatabase`
: The specified database is not a schema database.

`dmErrInvalidTableName`
: The database doesn't contain a table with the specified name.

Comments Column IDs are zero-based. That is, they range from zero to one less than the value returned by this function.

Example You can easily iterate through all of the columns in a table by doing something like this:

```
uint32_t numCols;
uint32_t idx;
uint32_t colID;

err = DbNumColumns(myDatabase, myTableName, &numCols);
for(idx = 0; idx < numCols; idx++){
   err = DbGetColumnID(myDatabase, myTableName, idx, &colID);
   // do something based upon colID here
}
```

See Also `DbGetAllColumnDefinitions()`, `DbGetColumnDefinitions()`, `DbGetColumnID()`

DbNumSortIndexes Function

Purpose Get the number of sort indices defined for a given database.

Declared In `SchemaDatabases.h`

Prototype
```
status_t DbNumSortIndexes (DmOpenRef dbRef,
   uint32_t *countP)
```

Parameters

→ *`dbRef`*
  `DmOpenRef` to an open database.

← *`countP`*
  The number of sort indices defined for the database.

Returns

`dmErrInvalidParam`
  `dbRef` doesn't reference an open database.

`dmErrNotSchemaDatabase`
  The specified database is not a schema database.

Comments This function returns the number of sort indices that are defined for a specified database. The index values of those sort indices range from 0 to one less than the value that this function returns. Most functions that take a sort index as an argument require the SQL statement used to create the sort index.

Example Code that iterates through all of the sort indices in a database might look something like this:

```
uint32_t numSortIndexes, idx;
char *sortIndex;

err = DbNumSortIndexes(myDatabase, &numSortIndexes);
if (err == errNone){
   for (idx = 0; idx < numSortIndexes; idx++){
      err = DbGetSortDefinition(myDatabase, idx, &sortIndex);
      if (err == errNone){
         // process sort index here. The SQL is in *sortIndex
      }
   }
}
```

See Also `DbGetSortDefinition()`, `DbHasSortIndex()`

DbNumTables Function

Purpose Get the number of tables defined for a given database.

Declared In `SchemaDatabases.h`

Prototype
```
status_t DbNumTables (DmOpenRef dbRef,
   uint32_t *tableCountP)
```

Parameters

→ *`dbRef`*
`DmOpenRef` to an open database.

← *`tableCountP`*
The number of schemas defined for the database.

Returns Returns `errNone` if no error, or one of the following if an error occurred:

`dmErrInvalidParam`
`dbRef` doesn't reference an open database.

`dmErrNotSchemaDatabase`
The specified database is not a schema database.

Comments This function returns the number of tables that a specified database contains. The indices of those tables range from 0 to one less than the value that this function returns. Most functions that take a table as an argument require the table's name.

Example Code that iterates through all of the tables in a database might look something like this:

```
uint32_t numTables, idx;
char tblName[dbDBNameLength];

err = DbNumTables(myDatabase, &numTables);
if (err == errNone){
   for (idx = 0; idx < numTables; idx++){
      err = DbGetTableName(myDatabase, idx, tblName);
      if (err == errNone){
         // process table here
      }
   }
}
```

See Also `DbGetTableName()`

DbOpenDatabase Function

Purpose Open a schema database and return a reference to it.

Declared In `SchemaDatabases.h`

Prototype
```
DmOpenRef DbOpenDatabase (DatabaseID dbID,
   DmOpenModeType mode, DbShareModeType share)
```

Parameters

→ *`dbID`*
The database ID of the schema database to be opened.

→ *`mode`*
Access mode with which to open the database. See `DmOpenModeType` for the set of values that you can supply for this parameter.

→ *`share`*
How the database can be accessed by other applications while your application has it open. See the definition of `DbShareModeType` for the set of values that you can supply for this parameter.

Returns A `DmOpenRef` to the open database. This function may display a fatal error message if *`dbID`* is `NULL`. For all other errors, this function returns 0; call `DmGetLastErr()` to obtain an error code indicating the reason for failure.

Comments The database must exist and either the application or the user—or both—must have correct access to open the database in the specified mode.

IMPORTANT: When called from the main application thread, this function may block. While blocked, the application will not receive events and won't redraw its windows. As well, deferred sublaunches and notifications won't execute while the main application thread is blocked.

See Also `DbCloseDatabase()`, `DbOpenDatabaseByName()`

DbOpenDatabaseByName Function

Purpose Open the most recent revision of a schema database with the given name and creator and return a reference to it.

Declared In `SchemaDatabases.h`

Prototype

```
DmOpenRef DbOpenDatabaseByName (uint32_t creator,
   const char *name, DmOpenModeType mode,
   DbShareModeType share)
```

Parameters

→ *creator*
Schema database creator.

→ *name*
Schema database type.

→ *mode*
Access mode with which to open the database. See `DmOpenModeType` for the set of values that you can supply for this parameter.

→ *share*
How the database can be accessed by other applications while your application has it open. See the definition of `DbShareModeType` for the set of values that you can supply for this parameter.

Returns A `DmOpenRef` to the open database. This function may display a fatal error message if *dbID* is `NULL`. For all other errors, this function returns 0; call `DmGetLastErr()` to obtain an error code indicating the reason for failure.

Comments The database must exist and either the application or the user—or both—must have correct access to open the database in the specified mode.

IMPORTANT: When called from the main application thread, this function may block. While blocked, the application will not receive events and won't redraw its windows. As well, deferred sublaunches and notifications won't execute while the main application thread is blocked.

See Also `DbCloseDatabase()`, `DbOpenDatabase()`

DbReleaseStorage Function

Purpose Release memory that was allocated by the operating system and returned to your application as the result of a function call such as `DbGetColumnValues()`.

Declared In `SchemaDatabases.h`

Prototype
```
status_t DbReleaseStorage (DmOpenRef dbRef,
   void *ptr)
```

Parameters

→ *`dbRef`*
: `DmOpenRef` to an open database.

→ *`ptr`*
: Pointer to the memory to be released. This block of memory must have been allocated by the operating system during the course of a call to one of the functions listed in the Comments section, below.

Returns Returns `errNone` if the operation completed successfully, or one of the following if an error occurred:

`dmErrInvalidParam`
: *`dbRef`* is `NULL` or *`ptr`* is `NULL`.

`dmErrNotSchemaDatabase`
: The specified database is not a schema database.

`dmErrReadOnly`
: The specified database is a read-only database or is open in read-only mode.

`dmErrCantFind`
: The block wasn't allocated by calling one of the functions listed in the Comments section, below.

`dmErrInvalidID`
: A column value cannot be freed because the ID of the row containing the value is invalid.

`dmErrUniqueIDNotFound`
: A column value cannot be freed because the row containing the value cannot be located.

`dmErrInvalidTableName`
: A column property value cannot be freed because the table name is no longer valid.

`dmErrInvalidColumnName`
A column property value cannot be freed because the name of the column is no longer valid.

`dmErrInvalidColumnID`
A column property value cannot be freed because the column's ID is no longer valid.

Comments Releases memory allocated by the following functions:

- `DbGetColumnValue()`
- `DbGetColumnValues()`
- `DbGetAllColumnValues()`
- `DbGetColumnPropertyValue()`
- `DbGetColumnPropertyValues()`
- `DbGetAllColumnPropertyValues()`
- `DbGetColumnDefinitions()`
- `DbGetAllColumnDefinitions()`

DbRemoveCategory Function

Purpose Remove membership in the specified categories from a single row.

Declared In `SchemaDatabases.h`

Prototype
```
status_t DbRemoveCategory (DmOpenRef dbRef,
   uint32_t rowID, uint32_t numToRemove,
   const CategoryID categoryIDs[])
```

Parameters

→ *`dbRef`*
`DmOpenRef` to an open database.

→ *`rowID`*
Row ID or cursor ID identifying the row for which category membership is to be altered.

→ *`numToRemove`*
Number of categories in the *`categoryIDs`* array.

→ *`categoryIDs`*
Array of category IDs indicating those categories for which the specified row is no longer to be a member.

Returns Returns `errNone` if the operation completed successfully, or one of the following if an error occurred:

`dmErrInvalidParam`
: *`dbRef`* doesn't reference an open database, *`rowID`* isn't a row or cursor ID, or *`numToRemove`* is nonzero and *`categoryIDs`* is `NULL`.

`dmErrNotSchemaDatabase`
: The specified database is not a schema database.

`dmErrReadOnly`
: The specified database is a read-only database or is open in read-only mode.

`dmErrUniqueIDNotFound`
: The specified row ID doesn't reference a row within the database.

`dmErrMemError`
: A memory error occurred.

`dmErrInvalidCategory`
: One or more of the specified categories is not a valid category.

`dmErrRecordBusy`
: The row is in use and cannot be updated.

Comments This function removes the specified category memberships from the specified row but does not remove the actual category definitions themselves, which are defined at the database level.

The database must be opened with write access. The specified category IDs must be valid.

This function ignores category IDs for which the specified row is not a member. If the *`categoryIDs`* array contains multiple instances of a given category ID, the category membership is removed when the first instance is encountered; the remaining instances are ignored.

See Also `DbAddCategory()`, `DbMoveCategory()`, `DbRemoveCategoryAllRows()`, `DbSetCategory()`

DbRemoveCategoryAllRows Function

Purpose Remove category membership in the specified categories from all rows in the database, depending on the match mode criteria.

Declared In `SchemaDatabases.h`

Prototype

```
status_t DbRemoveCategoryAllRows
    (DmOpenRef dbRef, uint32_t numCategories,
    const CategoryID categoryIDs[],
    DbMatchModeType matchMode)
```

Parameters

→ *`dbRef`*
`DmOpenRef` to an open database.

→ *`numCategories`*
Number of categories in the *`categoryIDs`* array.

→ *`categoryIDs`*
Array of category IDs indicating those categories for which the specified row is no longer to be a member.

→ *`matchMode`*
One of the following values:

`DbMatchAny`
(OR): Remove categories from rows matching any of the specified categories.

`DbMatchAll`
(AND): Remove categories from rows matching all of the specified categories, including rows with additional category membership.

`DbMatchExact`
Remove categories from rows matching exactly the specified categories.

Returns Returns `errNone` if the operation completed successfully, or one of the following if an error occurred:

`dmErrInvalidParam`
`dbRef` doesn't reference an open database, *`numCategories`* is zero and *`categoryIDs`* is not `NULL`, *`numCategories`* is nonzero and *`categoryIDs`* is `NULL`, or *`matchMode`* isn't one of the allowable values.

`dmErrNotSchemaDatabase`
: The specified database is not a schema database.

`dmErrReadOnly`
: The specified database is a read-only database or is open in read-only mode.

`dmErrMemError`
: A memory error occurred.

`dmErrInvalidCategory`
: One or more of the specified categories is not a valid category.

`dmErrRecordBusy`
: At least one of the database's rows is in use and cannot be updated.

Comments This function removes the specified category memberships from the specified row but does not remove the actual category definitions themselves, which are defined at the database level.

The database must be opened with write access. The specified category IDs must be valid.

This function might perform no action if

- none of the supplied category IDs are valid and the match mode is `DbMatchAny`.
- any of the category IDs are not valid and the match mode is either `DbMatchAll` or `DbMatchExact`.

See Also `DbAddCategory()`, `DbRemoveCategory()`, `DbSetCategory()`

DbRemoveColumn Function

Purpose Remove a column definition from a specified database schema and remove that column's data for all table rows described by that schema.

Declared In `SchemaDatabases.h`

Prototype
```
status_t DbRemoveColumn (DmOpenRef dbRef,
    const char *table, uint32_t columnID)
```

Parameters → *`dbRef`*
: `DmOpenRef` to an open database.

→ *`table`*
Table name.

→ *`columnID`*
ID of the column being removed.

Returns Returns `errNone` if the operation completed successfully, or one of the following if an error occurred:

`dmErrInvalidParam`
`dbRef` doesn't reference an open database, or *`table`* is `NULL`.

`dmErrNotSchemaDatabase`
The specified database is not a schema database.

`dmErrReadOnly`
The database is not open for write access.

`dmErrAccessDenied`
You do not have authorization to modify the schema.

`dmErrInvalidTableName`
The database doesn't contain a table with the specified name.

`dmErrColumnDefinitionsLocked`
The table's column definitions are locked.

`dmErrInvalidColSpec`
The table has no columns defined.

`dmErrInvalidColumnID`
The specified table doesn't have a column with the supplied column ID.

`dmErrRecordBusy`
One or more rows are in use and cannot be modified.

`dmErrMemError`
A memory error occurred.

See Also `DbAddColumn()`, `DbRemoveColumnProperty()`

DbRemoveColumnProperty Function

Purpose Remove a single column property from a database table.

Declared In `SchemaDatabases.h`

Prototype
```
status_t DbRemoveColumnProperty (DmOpenRef dbRef,
   const char *table, uint32_t columnID,
   DbSchemaColumnProperty propID)
```

Parameters

→ *`dbRef`*
  `DmOpenRef` to an open database.

→ *`table`*
  Table name.

→ *`columnID`*
  ID of the column for which the property is being removed.

→ *`propID`*
  ID of the column property being removed.

Returns Returns `errNone` if the operation completed successfully, or one of the following if an error occurred:

`dmErrInvalidParam`
  `dbRef` doesn't reference an open database, or *`table`* is `NULL`.

`dmErrBuiltInProperty`
  The column property you are trying to remove is a built-in property; it cannot be removed.

`dmErrNotSchemaDatabase`
  The specified database is not a schema database.

`dmErrReadOnly`
  The database is not open for write access.

`dmErrAccessDenied`
  You do not have authorization to modify the schema.

`dmErrInvalidTableName`
  The database doesn't contain a table with the specified name.

`dmErrColumnDefinitionsLocked`
  The table's column definitions are locked.

`dmErrInvalidColSpec`
  The table has no columns defined.

`dmErrInvalidColumnID`
The specified table doesn't have a column with the supplied column ID.

`dmErrColumnPropertiesLocked`
The specified column property is locked.

`dmErrInvalidPropID`
The specified column property ID doesn't reference a column within the table.

Comments This function removes the property corresponding to *`propID`*. The memory associated with the property value is freed.

See Also `DbSetColumnPropertyValue()`, `DbRemoveColumn()`

DbRemoveRow Function

Purpose Remove a row from a database and dispose of its data chunks.

Declared In `SchemaDatabases.h`

Prototype
```
status_t DbRemoveRow (DmOpenRef dbRef,
   uint32_t rowID)
```

Parameters

→ *`dbRef`*
`DmOpenRef` to an open database.

→ *`rowID`*
Row ID or cursor ID identifying the row being removed.

Returns Returns `errNone` if the row was successfully removed, or one of the following if an error occurred:

`dmErrInvalidParam`
`dbRef` doesn't reference an open database, *`rowID`* isn't a cursor or row ID, or *`rowID`* is a cursor ID but doesn't represent a valid row within the cursor.

`dmErrNotSchemaDatabase`
The specified database is not a schema database.

`dmErrReadOnly`
The database is not open for write access.

`dmErrRecordBusy`
The specified row is in use and cannot be removed.

`memErrNotEnoughSpace`
A memory error occurred.

See Also `DbArchiveRow()`, `DbCursorRemoveAllRows()`, `DbDeleteRow()`, `DbInsertRow()`, `DbRemoveSecretRows()`

DbRemoveSecretRows Function

Purpose Remove all secret rows from the database.

Declared In `SchemaDatabases.h`

Prototype `status_t DbRemoveSecretRows (DmOpenRef` *`dbRef`*`)`

Parameters → *`dbRef`*
`DmOpenRef` to an open database.

Returns Returns `errNone` if the operation completed successfully, or one of the following if an error occurred:

`dmErrInvalidParam`
`dbRef` doesn't reference an open database.

`dmErrNotSchemaDatabase`
The specified database is not a schema database.

`dmErrReadOnly`
The database is not open for write access.

`dmErrRecordBusy`
At least one of the database's secret rows is in use and cannot be removed.

`memErrNotEnoughSpace`
A memory error occurred.

See Also `DbRemoveRow()`

DbRemoveSortIndex Function

Purpose Remove a sort index from a database.

Declared In `SchemaDatabases.h`

Prototype
```
status_t DbRemoveSortIndex (DmOpenRef dbRef,
   const char *orderBy)
```

Parameters

→ *dbRef*
: `DmOpenRef` to an open database.

→ *orderBy*
: The sort index to be removed. See "The SELECT Statement" on page 37 for the format of this parameter.

Returns Returns `errNone` if the operation completed successfully, or one of the following if an error occurred:

`dmErrInvalidParam`
: *dbRef* doesn't reference an open database.

`dmErrNotSchemaDatabase`
: The specified database is not a schema database.

`dmErrReadOnly`
: The database is not open for write access.

`dmErrAccessDenied`
: You do not have authorization to modify the database.

`dmErrSQLParseError`
: The specified table name or the sort information specified in the sort index is invalid.

`dmErrInvalidSortDefn`
: The specified sort index isn't defined for this database.

`dmErrMemError`
: A memory error occurred.

Comments The database must exist and the application or user—or both—must have write authorization to the database. The specified sort index must also exist.

See Also `DbAddSortIndex()`, `DbHasSortIndex()`

DbRemoveTable Function

Purpose Remove a table definition from a schema database.

Declared In `SchemaDatabases.h`

Prototype
```
status_t DbRemoveTable (DmOpenRef dbRef,
   const char *table)
```

Parameters

→ *`dbRef`*
: `DmOpenRef` to an open database.

→ *`table`*
: Table name.

Returns Returns `errNone` if the operation completed successfully, or one of the following if an error occurred:

`dmErrInvalidParam`
: *`dbRef`* doesn't reference an open database, or *`table`* is `NULL`.

`dmErrNotSchemaDatabase`
: The specified database is not a schema database.

`dmErrInvalidTableName`
: *`table`* is not the name of a table in the specified database.

`dmErrReadOnly`
: You've attempted to write to or modify a database that is open in read-only mode.

`dmErrAccessDenied`
: You do not have authorization to modify the database, or one or more sort indices are defined for the table.

`dmErrTableNotEmpty`
: The table contains one or more non-deleted rows.

Comments You cannot remove a table if it contains one or more non-deleted rows or if any sort indices are defined for the table. You must first delete or remove any such rows and sort indices before you can remove the table.

See Also `DbAddTable()`

DbSetCategory Function

Purpose Set category membership for a single database row.

Declared In `SchemaDatabases.h`

Prototype
```
status_t DbSetCategory (DmOpenRef dbRef,
   uint32_t rowID, uint32_t numToSet,
   const CategoryID categoryIDs[])
```

Parameters

→ *`dbRef`*
: `DmOpenRef` to an open database.

→ *`rowID`*
: Row ID or cursor ID identifying the row for which category membership is being set.

→ *`numToSet`*
: Number of category IDs in the *`categoryIDs`* array.

→ *`categoryIDs`*
: Array of category IDs identifying the categories that the row is to be a member of. Upon successful completion, the row is a member only of those categories identified in this array.

Returns Returns `errNone` if the operation completed successfully, or one of the following if an error occurred:

`dmErrInvalidParam`
: *`dbRef`* doesn't reference an open database, the specified row or cursor ID is not valid, or *`numToASet`* is nonzero and *`categoryIDs`* is `NULL`.

`dmErrNotSchemaDatabase`
: The specified database is not a schema database.

`dmErrReadOnly`
: The database is not open for write access.

`dmErrIndexOutOfRange`
: The specified row or cursor ID doesn't reference a row within the table.

`dmErrRecordDeleted`
: The specified row is marked as deleted.

`dmErrRecordBusy`
: The specified row is in use and cannot be updated.

`dmErrMemError`
A memory error occurred.

`dmErrInvalidCategory`
The allowed number of categories has been exceeded, or a category ID doesn't correspond to a defined category.

Comments Any previous category membership for the row is overwritten by the specified category membership. To remove all category membership from a row (making it "Unfiled"), set *`numToSet`* to 0 and *`categoryIDs`* to `NULL`.

The database must be opened with write access. The supplied category IDs must be valid.

If a given category ID occurs more than once in the category ID array, the row is made a member of the category and the duplicate category IDs are ignored.

See Also `DbAddCategory()`, `DbGetCategory()`, `DbRemoveCategory()`

DbSetColumnPropertyValue Function

Purpose Set a single property value for a database column property.

Declared In `SchemaDatabases.h`

Prototype
```
status_t DbSetColumnPropertyValue
   (DmOpenRef dbRef, const char *table,
   uint32_t columnID,
   DbSchemaColumnProperty propID,
   uint32_t numBytes, const void *propValueP)
```

Parameters

→ *`dbRef`*
`DmOpenRef` to an open database.

→ *`table`*
Table name.

→ *`columnID`*
ID of the column for which the property value is being set.

→ *`propID`*
ID of the property being set.

→ *`numBytes`*
Size, in bytes, of the property value.

→ *`propValueP`*
The property value.

Returns Returns `errNone` if the property value was successfully set, or one of the following otherwise:

`dmErrInvalidParam`
`dbRef` doesn't reference an open database, *`table`* is `NULL`, or *`numBytes`* is nonzero and *`propValueP`* is `NULL`.

`dmErrBuiltInProperty`
The specified property is a built-in property.

`dmErrNotSchemaDatabase`
The specified database is not a schema database.

`dmErrReadOnly`
The database is not open for write access.

`dmErrAccessDenied`
You are not authorized to write to this table.

`dmErrInvalidTableName`
`table` isn't defined within this database.

`dmErrInvalidColumnID`
The table doesn't have a column with the specified column ID.

`dmErrColumnPropertiesLocked`
The specified column property is locked.

`dmErrMemError`
A memory error occurred.

`memErrNotEnoughSpace`
A memory error occurred.

Comments This function frees the existing column property value and copies the supplied property value to the storage heap. Because it makes a copy of the property value, after calling this function your application can free any local copy of the property value.

See Also `DbGetColumnPropertyValue()`, `DbSetColumnPropertyValues()`

DbSetColumnPropertyValues Function

Purpose Set one or more database column property values.

Declared In `SchemaDatabases.h`

Prototype
```
status_t DbSetColumnPropertyValues
   (DmOpenRef dbRef, const char *table,
   uint32_t numProps,
   const DbColumnPropertyValueType propValues[])
```

Parameters

→ *`dbRef`*
: `DmOpenRef` to an open database.

→ *`table`*
: Table name.

→ *`numProps`*
: Number of elements in the *`propValues`* array.

→ *`propValues`*
: Array of structures, each of which identifies a column, a property, and a property value. See `DbColumnPropertyValueType` for a description of the structure.

Returns Returns `errNone` if the property value was successfully set, or one of the following otherwise:

`dmErrInvalidParam`
: *`dbRef`* doesn't reference an open database, *`table`* is `NULL`, *`numProps`* is nonzero, or *`propValues`* is `NULL`.

`dmErrNotSchemaDatabase`
: The specified database is not a schema database.

`dmErrReadOnly`
: The database is not open for write access.

`dmErrAccessDenied`
: You are not authorized to write to this table.

`dmErrInvalidTableName`
: *`table`* isn't defined within this database.

`dmErrInvalidColumnID`
: One of the specified column IDs doesn't correspond to a table column.

`dmErrColumnPropertiesLocked`
: One of the column properties is locked.

`dmErrMemError`
: A memory error occurred.

`memErrNotEnoughSpace`
: A memory error occurred.

Comments This function creates a column property if it does not exist and frees an existing column property value if the column property already exists. It copies the supplied property values to the storage heap. Because it makes a copy of each supplied property value, after calling this function your application can free any local copies of the property values.

See Also `DbGetColumnPropertyValues()`, `DbSetColumnPropertyValue()`

DbSetRowAttr Function

Purpose Set the attributes of a row.

Declared In `SchemaDatabases.h`

Prototype

```
status_t DbSetRowAttr (DmOpenRef dbRef,
   uint32_t rowID, uint16_t *attrP)
```

Parameters

→ *`dbRef`*
: `DmOpenRef` to an open database.

→ *`rowID`*
: Row ID or cursor ID identifying the row for which attributes are being set.

→ *`attrP`*
: Pointer to the new attributes for the row.

Returns Returns `errNone` if the attributes were set successfully, or one of the following if an error occurred:

`dmErrNotRecordDB`
: You've attempted to perform a row function on a resource database.

`dmErrIndexOutOfRange`
: The specified index is out of range.

`dmErrReadOnly`
You've attempted to write to or modify a database that is open in read-only mode.

Comments Row attributes are documented under "Schema Database Row Attributes" on page 300. This function can be used only to set those attributes that are not system-only attributes (system-only attributes are those that make up `dbSysOnlyRecAttrs`).

See Also `DbGetRowAttr()`

DbWriteColumnValue Function

Purpose Write a single column value for a row.

Declared In `SchemaDatabases.h`

Prototype

```
status_t DbWriteColumnValue (DmOpenRef dbRef,
   uint32_t rowID, uint32_t columnID,
   uint32_t offset, int32_t bytesToReplace,
   const void *srcP, uint32_t srcBytes)
```

Parameters

→ *`dbRef`*
`DmOpenRef` to an open database.

→ *`rowID`*
The row ID or cursor ID identifying the row for which the column value is being written.

→ *`columnID`*
ID of the column being written.

→ *`offset`*
For variable-length columns, an offset, in bytes, to the location within the column where the value is to be written.

→ *`bytesToReplace`*
For variable-length columns, the number of data bytes to be replaced by the write operation, or -1 to replace all of the column's data for the row.

→ *`srcP`*
Data to write into the column.

→ *`srcBytes`*
Number of bytes to write.

Returns Returns `errNone` if the data was successfully written, or one of the following otherwise:

`dmErrInvalidParam`
: *`dbP`* doesn't reference an open database, or *`rowID`* isn't a row or cursor ID.

`dmErrCursorBOF`
: The specified cursor ID is BOF.

`dmErrCursorEOF`
: The specified cursor ID is EOF.

`dmErrUniqueIDNotFound`
: The specified row ID doesn't correspond to a valid row within the table.

`dmErrNotSchemaDatabase`
: The specified database is not a schema database.

`dmErrReadOnly`
: The database is not open for write access.

`dmErrRecordDeleted`
: The row is marked as deleted.

`dmErrRecordBusy`
: The row is busy and cannot be written to.

`dmErrInvalidTableName`
: The database doesn't contain a table with the specified name.

`dmErrMemError`
: A memory error occurred.

`dmErrWriteOutOfBounds`
: The write exceeded the bounds of the column.

`memErrNotEnoughSpace`
: A memory error occurred.

Comments To remove existing column data, set *`srcP`* to `NULL`. If *`srcP`* is `NULL`, *`srcBytes`* is ignored.

Offset-based writes are not supported for fixed-length column data types; the *`offset`* and *`bytesToReplace`* parameters are ignored for them. The list of column data types supporting offset based writes are:

- `VarChar`

- `Blob`
- `Vector`

`DbWriteColumnValue()` does not merely replace one set of bytes with an equal-sized set; depending on the *`bytesToReplace`* and *`srcBytes`* parameters, the resulting value can be shorter or longer than the original value. The following sections detail the operations you can perform with this function.

Expand

If *`bytesToReplace`* is less than *`srcBytes`*, the resulting column value is longer than the original value. For instance:

Original column data: `"abcde"`

`offset`: 2

`bytesToReplace`: 2

`srcBytes`: 8

**`srcP`*: `"12345678"`

Updated column data: `"ab12345678e"`

Shrink

If *`bytesToReplace`* is greater than *`srcBytes`*, the resulting column value is shorter than the original value. For instance:

Original column data: `"abcde"`

`offset`: 2

`bytesToReplace`: 3

`srcBytes`: 1

**`srcP`*: `"1"`

Updated column data: `"ab1"`

Truncate

Taking the "shrink" scenario to its extreme, to simply remove a portion of the original column data, set *`srcBytes`* to 0, as shown here:

Original column data: `"abcde"`

`offset`: 2

`bytesToReplace`: 3

`srcBytes`: 0

*`srcP`: `NULL`

Updated column data: `"ab"`

Insert

If `bytesToReplace` is 0, the data is inserted into the original column data. For instance:

Original column data: `"abcde"`

`offset`: 2

`bytesToReplace`: 0

`srcBytes`: 5

*`srcP`: `"12345"`

Updated column data: `"ab12345cde"`

Append

A variant on the “insert” scenario, if the `offset` parameter is set to the length of the current column data and `bytesToReplace` is 0, the data being written is appended to the current column data. For example:

Original column data: `"abcde"`

`offset`: 5

`bytesToReplace`: 0

`srcBytes`: 5

*`srcP`: `"12345"`

Updated column data: `"abcde12345"`

Partial Replacement

To replace a portion of the original column data without changing the size of the column data, `bytesToReplace` should equal `srcBytes`, as shown here:

Original column data: `"abcde"`

`offset`: 2

`bytesToReplace`: 2

`srcBytes`: 2

**`srcP`*: `"12"`

Updated column data: `"ab12e"`

Complete Replacement

To completely replace a column's data, set *`offset`* to 0 and *`bytesToReplace`* to -1. For example:

Original column data: `"abcde"`

`offset`: 0

`bytesToReplace`: -1

`srcBytes`: 5

**`srcP`*: `"12345"`

Updated column data: `"12345"`

See Also `DbCopyColumnValue()`, `DbGetColumnValue()`, `DbWriteColumnValues()`

DbWriteColumnValues Function

Purpose Write one or more column values for a row.

Declared In `SchemaDatabases.h`

Prototype

```
status_t DbWriteColumnValues (DmOpenRef dbRef,
   uint32_t rowID, uint32_t numColumnValues,
   DbSchemaColumnValueType *columnValuesP)
```

Parameters

→ *`dbRef`*
`DmOpenRef` to an open database.

→ *`rowID`*
The row ID or cursor ID identifying the row for which the column values are being written.

→ *`numColumnValues`*
Number of elements in the *`columnValuesP`* array.

→ *`columnValuesP`*
Array of structures, each containing a column ID and a value.

Returns Returns `errNone` if the data was successfully written, or one of the following otherwise:

`dmErrInvalidParam`
`dbP` doesn't reference an open database, or *`rowID`* isn't a row or cursor ID.

`dmErrCursorBOF`
The specified cursor ID is BOF.

`dmErrCursorEOF`
The specified cursor ID is EOF.

`dmErrUniqueIDNotFound`
The specified row ID doesn't correspond to a valid row within the table.

`dmErrNotSchemaDatabase`
The specified database is not a schema database.

`dmErrReadOnly`
The database is not open for write access.

`dmErrRecordDeleted`
The row is marked as deleted.

`dmErrRecordBusy`
The row is busy and cannot be written to.

`dmErrInvalidTableName`
The database doesn't contain a table with the specified name.

`dmErrMemError`
A memory error occurred.

`dmErrWriteOutOfBounds`
The write exceeded the bounds of a column.

`memErrNotEnoughSpace`
A memory error occurred.

Comments A `NULL` value for the `data` field of the `DbSchemaColumnValueType` structure is allowed; this removes existing column data for the specified column and row.

See Also `DbCopyColumnValues()`, `DbGetColumnValues()`, `DbWriteColumnValue()`

8

VFS Manager

The Virtual File System (VFS) Manager is a layer of software that manages all installed file system libraries. It provides a unified API to application 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 Compact Flash, Memory Stick, and SmartMedia.This chapter provides reference material for the VFS Manager API, organized as follows:

The header file `VFSMgr.h` declares the API that this chapter describes.

For more information on file systems in Palm OS® and the VFS Manager, see Chapter 3, “Virtual File Systems,” on page 69.

VFS Manager Structures and Types

FileInfoType Struct

Purpose Contains information about a specified file or directory.

Declared In `VFSMgr.h`

Prototype

```
typedef struct FileInfoTag {
   uint32_t attributes;
   char *nameP;
   uint16_t nameBufLen;
   uint16_t reserved;
} FileInfoType, *FileInfoPtr
```

Fields

`attributes`
Characteristics of the file or directory. See File and Directory Attributes for the bits that make up this field.

`nameP`
Pointer to the buffer that receives the full name of the file or directory. Initialize this parameter to `NULL` if you don't want to receive the name.

`nameBufLen`
Size of the `nameP` buffer, in bytes.

`reserved`
Reserved for future use.

Comments This information is returned as a parameter to `VFSDirEntryEnumerate()`.

FileOrigin Typedef

Purpose Encodes references to files and directories.

Declared In `VFSMgr.h`

Prototype `typedef uint16_t FileOrigin`

FileRef Typedef

Purpose Container for a reference to an opened file or directory which is supplied to various `VFSFile...` operations.

Declared In `VFSMgr.h`

Prototype `typedef uint32_t FileRef`

Comments Use `VFSFileOpen()` to obtain a `FileRef` value.

VFSAnyMountParamType Struct

Purpose A base structure for `VFSSlotMountParamType`, `VFSPOSEMountParamType`, and other similar structures that may be defined in the future. Use one or the other according to how you set the `mountClass` parameter.

Declared In `VFSMgr.h`

Prototype

```
typedef struct VFSAnyMountParamTag {
   uint16_t volRefNum;
   uint16_t size;
   uint32_t mountClass;
} VFSAnyMountParamType
typedef VFSAnyMountParamType *VFSAnyMountParamPtr
```

Fields

`volRefNum`
The volume reference number. This is initially obtained when you successfully mount a volume. It can then be used to format a volume with `VFSVolumeFormat()` or unmount a volume with `VFSVolumeUnmount()`.

`size`

`mountClass`
Defines the type of mount to use with the specified volume. See Volume Mount Classes for a list of mount types.

VFSPOSEMountParamType Struct

Purpose When you are mounting a volume through Palm OS® Emulator, the `vfsMountParam->mountClass` must be set to `VFSMountClass_POSE`. Note that ordinary applications and file systems shouldn't use `VFSPOSEMountParamType`.

Declared In `VFSMgr.h`

Prototype

```
typedef struct VFSPOSEMountParamTag {
   VFSAnyMountParamType vfsMountParam;
   uint8_t poseSlotNum;
   uint8_t reserved;
   uint16_t reserved2;
} VFSPOSEMountParamType
```

Fields

`vfsMountParam`
: See the description of `VFSAnyMountParamType` for an explanation of the fields in this structure. Set `vfsMountParam->mountClass` to `VFSMountClass_POSE` to mount a virtual slot.

`poseSlotNum`
: Number of the virtual slot number to be mounted by Palm OS Emulator.

`reserved`
: Reserved for future use.

`reserved2`
: Reserved for future use.

VFSSlotMountParamType Struct

Purpose When you are mounting a card located in an Expansion Manager slot, the `vfsMountParam->mountClass` field must be set to `VFSMountClass_SlotDriver`.

Declared In `VFSMgr.h`

Prototype

```
typedef struct VFSSlotMountParamTag {
   VFSAnyMountParamType vfsMountParam;
   uint16_t slotLibRefNum;
   uint16_t slotRefNum;
} VFSSlotMountParamType
```

Fields

`vfsMountParam`
: See the description of `VFSAnyMountParamType` for an explanation of the fields in this structure. Set `vfsMountParam->mountClass` to `VFSMountClass_SlotDriver` to mount an Expansion Manager slot.

`slotLibRefNum`
: Reference number for the slot driver library allocated to the given slot number.

`slotRefNum`
: Number of the slot to be mounted.

VolumeInfoType Struct

Purpose Define information that is returned to `VFSVolumeInfo()` and used throughout the VFS functions.

Declared In `VFSMgr.h`

Prototype

```
typedef struct VolumeInfoTag {
   uint32_t attributes;
   uint32_t fsType;
   uint32_t fsCreator;
   uint32_t mountClass;
   uint16_t slotLibRefNum;
   uint16_t slotRefNum;
   uint32_t mediaType;
   uint32_t reserved;
} VolumeInfoType, *VolumeInfoPtr
```

Fields

`attributes`
Characteristics of the volume. See Volume Attributes for the bits that make up this field.

`fsType`
File system type for this volume. See Defined File Systems for a list of the supported file systems.

`fsCreator`
Creator ID of this volume's file system driver. This information is used with `VFSCustomControl()`.

`mountClass`
Mount class that mounted this volume. The supported mount classes are listed under Volume Mount Classes.

`slotLibRefNum`
Reference to the slot driver library with which the volume is mounted. This field is only valid when the mount class is `vfsMountClass_SlotDriver`.

`slotRefNum`
Slot number where the card containing the volume is loaded. This field is only valid when the mount class is `vfsMountClass_SlotDriver`.

`mediaType`
Type of card media. See Defined Media Types in Chapter 25, "Expansion Manager," of *Exploring Palm OS: System*

Management for the list of values. This field is only valid when the mount class is `vfsMountClass_SlotDriver`.

`reserved`
Reserved for future use.

VFS Manager Constants

VFS Manager Error Codes

Purpose Error codes returned by the various VFS Manager functions.

Declared In `VFSMgr.h`

Constants

`#define vfsErrBadData (vfsErrorClass | 12)`
The operation could not be completed because of invalid data.

`#define vfsErrBadName (vfsErrorClass | 14)`
Invalid filename, path, or volume label.

`#define vfsErrBufferOverflow (vfsErrorClass | 1)`
The supplied buffer is too small.

`#define vfsErrDirectoryNotFound (vfsErrorClass | 19)`
Returned when the path leading up to the file does not exist.

`#define vfsErrDirNotEmpty (vfsErrorClass | 13)`
The directory is not empty and therefore cannot be deleted.

`#define vfsErrFileAlreadyExists (vfsErrorClass | 6)`
A file with this name exists already in this location.

`#define vfsErrFileBadRef (vfsErrorClass | 3)`
The file reference is invalid: it has been closed or was not obtained from `VFSFileOpen()`.

`#define vfsErrFileEOF (vfsErrorClass | 7)`
The file pointer is at the end of the file.

`#define vfsErrFileGeneric (vfsErrorClass | 2)`
Generic file error.

`#define vfsErrFileNotFound (vfsErrorClass | 8)`
The file was not found at the specified location.

`#define vfsErrFilePermissionDenied (vfsErrorClass | 5)`
The requested permissions could not be granted.

`#define vfsErrFileStillOpen (vfsErrorClass | 4)`
Returned from the underlying file system's delete function if the file is still open.

`#define vfsErrIsADirectory (vfsErrorClass | 18)`
This operation can only be performed on a regular file, not a directory.

`#define vfsErrNameShortened (vfsErrorClass | 20)`
A volume name or filename was automatically shortened to conform to the file system specification.

`#define vfsErrNoFileSystem (vfsErrorClass | 11)`
None of the installed file systems support this operation.

`#define vfsErrNotADirectory (vfsErrorClass | 17)`
This operation can only performed on a directory.

`#define vfsErrUnimplemented (vfsErrorClass | 16)`

`#define vfsErrVolumeBadRef (vfsErrorClass | 9)`
The volume reference number is invalid.

`#define vfsErrVolumeFull (vfsErrorClass | 15)`
There is insufficient space left on the volume.

`#define vfsErrVolumeStillMounted (vfsErrorClass | 10)`
Returned from the underlying file system's format function if the volume is still mounted.

Defined File Systems

Purpose Identifiers for those file systems that are currently defined by the VFS Manager. These values are used with `VFSVolumeInfo()` in the `VolumeInfoType.fsType` parameter.

Declared In `VFSMgr.h`

Constants

`#define vfsFilesystemType_AFS 'afsu'`
Unix Andrew file system

`#define vfsFilesystemType_EXT2 'ext2'`
Linux file system

`#define vfsFilesystemType_FAT 'fats'`
FAT32, FAT16, and FAT12, but only using 8.3 filenames

`#define vfsFilesystemType_FFS 'ffsb'`
Unix Berkeley block based file system

`#define vfsFilesystemType_HFS 'hfss'`
Macintosh standard hierarchical file system

`#define vfsFilesystemType_HFSPlus 'hfse'`
Macintosh extended hierarchical file system

`#define vfsFilesystemType_HPFS 'hpfs'`
OS2 High Performance file system

`#define vfsFilesystemType_MFS 'mfso'`
Macintosh original file system

`#define vfsFilesystemType_NFS 'nfsu'`
Unix Networked file system

`#define vfsFilesystemType_Novell 'novl'`
Novell file system

`#define vfsFilesystemType_NTFS 'ntfs'`
Windows NT file system

`#define vfsFilesystemType_VFAT 'vfat'`
FAT32, FAT16, and FAT12 extended to handle long filenames

Open Mode Constants

Purpose Modes in which a file or directory is opened. They are used for the `openMode` parameter to the `VFSFileOpen()` function.

Declared In `VFSMgr.h`

Constants

```
#define vfsModeAll (vfsModeExclusive | vfsModeRead
   | vfsModeWrite | vfsModeCreate | vfsModeTruncate
   | vfsModeReadWrite | vfsModeLeaveOpen)
```
The complete set of open modes.

```
#define vfsModeCreate (0x0008U)
```
Create the file if it doesn't already exist. This open mode is implemented in the VFS layer, rather than in the file system library.

```
#define vfsModeExclusive (0x0001U)
```
Open and lock the file or directory. This mode excludes anyone else from using the file or directory until it is closed.

```
#define vfsModeLeaveOpen (0x0020U)
```
Leave the file open even after the application exits.

```
#define vfsModeRead (0x0002U)
```
Open for read access.

```
#define vfsModeReadWrite (vfsModeWrite |
  vfsModeRead)
```
Open for read/write access.

```
#define vfsModeTruncate (0x0010U)
```
Truncate the file to zero (0) bytes after opening, removing all existing data. This open mode is implemented in the VFS layer, rather than in the file system library.

```
#define vfsModeVFSLayerOnly (vfsModeCreate |
  vfsModeTruncate)
```
Mask used to isolate those flags that are only used by the VFS layer. These flags are not passed to the file system layer.

```
#define vfsModeWrite (0x0004U | vfsModeExclusive)
```
Open for write access.

File and Directory Attributes

Purpose 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.

Declared In `VFSMgr.h`

Constants

`#define vfsFileAttrAll (0x0000007fUL)`
The complete set of file and directory attributes.

`#define vfsFileAttrArchive (0x00000020UL)`
Archived file or directory

`#define vfsFileAttrDirectory (0x00000010UL)`
Directory

`#define vfsFileAttrHidden (0x00000002UL)`
Hidden file or directory

`#define vfsFileAttrLink (0x00000040UL)`
Link to another file or directory

`#define vfsFileAttrReadOnly (0x00000001UL)`
Read-only file or directory

`#define vfsFileAttrSystem (0x00000004UL)`
System file or directory

`#define vfsFileAttrVolumeLabel (0x00000008UL)`
Volume label

Volume Attributes

Purpose Bits that can be used individually or in combination to make up the attributes field in the `VolumeInfoType` structure.

Declared In `VFSMgr.h`

Constants

`#define vfsVolumeAttrHidden (0x00000004UL)`
The volume should not be visible to the user.

`#define vfsVolumeAttrReadOnly (0x00000002UL)`
The volume is read only.

`#define vfsVolumeAttrSlotBased (0x00000001UL)`
Reserved. Check the mount class to determine how a volume is mounted.

Volume Mount Classes

Purpose Define how a given volume is mounted. The `mountClass` field in the `VFSAnyMountParamType` and `VolumeInfoType` structures takes on one of these values.

Declared In `VFSMgr.h`

Constants `#define vfsMountClass_POSE 'pose'`
Mount the volume through Palm OS Emulator. This is used for testing.

`#define vfsMountClass_POSE_BE 'esop'`
Mount the volume through Palm OS Emulator, using big-endan ordering. This is used for testing.

`#define vfsMountClass_SlotDriver sysFileTSlotDriver`
Mount the volume with a slot driver shared library.

`#define vfsMountClass_SlotDriver_BE 'sbil'`
Mount the volume with a slot driver shared library, using big-endian ordering.

Date Types

Purpose Dates that can be obtained for an open file or directory.

Declared In `VFSMgr.h`

Constants `#define vfsFileDateAccessed (3)`
Date the file was last accessed.

`#define vfsFileDateCreated (1)`
File creation date.

`#define vfsFileDateModified (2)`
Date the file was last modified.

Comments Use `VFSFileGetDate()` to obtain these dates for an open file or directory, and `VFSFileSetDate()` to set them.

Seek Origins

Purpose File positions to which an offset is added (or subtracted, if the offset is negative) to get a seek position within the file.

Declared In `VFSMgr.h`

Compatibility `#define vfsOriginBeginning (0)`
The beginning of the file.

`#define vfsOriginCurrent (1)`
The current position within the file.

`#define vfsOriginEnd (2)`
The end of the file. Only negative offsets are allowed when `origin` is set to `vfsOriginEnd`.

Iterator Controls and Constants

Purpose Control the directory and volume iteration process.

Declared In `VFSMgr.h`

Constants `#define vfsIteratorStart (0L)`
Start iterating.

`#define vfsIteratorStop (0xffffffffL)`
Iteration is complete.

`#define vfsInvalidFileRef (0L)`
There are no more files to be enumerated or an error occurred.

`#define vfsInvalidVolRef (0)`
There are no more volumes to be enumerated or an error occurred.

Comments To iterate the contents of a directory, use `VFSDirEntryEnumerate()`. To iterate the contents of a volume, use `VFSVolumeEnumerate()`.

Volume Mount Flags

Purpose Flags that control how a volume is mounted.

Declared In `VFSMgr.h`

Constants

`#define vfsMountFlagsReserved1 (0x08)`
Reserved for future use.

`#define vfsMountFlagsReserved2 (0x10)`
Reserved for future use.

`#define vfsMountFlagsReserved3 (0x20)`
Reserved for future use.

`#define vfsMountFlagsReserved4 (0x40)`
Reserved for future use.

`#define vfsMountFlagsReserved5 (0x80)`
Reserved for future use.

`#define vfsMountFlagsUseThisFileSystem (0x01)`
Pass this flag to cause the volume to be mounted or formatted using the file system specified by the specified file system.

Comments Volumes can be mounted explicitly, with `VFSVolumeMount()`, or as part of the volume format process, done with `VFSVolumeFormat()`.

Pass no flags (0) to have the VFS Manager attempt to mount or format the volume using a file system appropriate to the slot.

Miscellaneous Constants and Definitions

Purpose The VFS Manager also includes these `#defines`.

Declared In `VFSMgr.h`

Constants

`#define SIZEOF_LargestVFSMountParamType (128)`

`#define SIZEOF_VFSAnyMountParamType (8)`

```
#define SIZEOF_VFSPOSEMountParamType
  (SIZEOF_VFSAnyMountParamType + 4)
```

```
#define SIZEOF_VFSSlotMountParamType
  (SIZEOF_VFSAnyMountParamType + 4)
```

`#define vfsFtrIDDefaultFS (1)`
: Feature number used in conjunction with a creator ID of `sysFileCVFSMgr` to determine the device's default filesystem.

`#define vfsFtrIDVersion (0)`
: Feature number used to obtain the version of the VFS Manager in the device's ROM. Use this number in conjunction with a creator ID of `sysFileCVFSMgr`.

`#define vfsHandledStartPrc (0x02)`

`#define vfsHandledUIAppSwitch (0x01)`

`#define vfsMgrVersionNum ((uint16_t)300)`
: The version of the VFS Manager APIs in this SDK. Compare this to the value of the `vfsFtrIDVersion` feature.

VFS Manager Functions and Macros

VFSCustomControl Function

Purpose Make a custom API call to a particular file system, given its creator ID. You can use `VFSVolumeInfo()` to determine the creator ID of the file system for a given volume.

Declared In `VFSMgr.h`

Prototype

```
status_t VFSCustomControl (uint32_t fsCreator,
   uint32_t apiCreator, uint16_t apiSelector,
   void *valueP, uint16_t *valueLenP)
```

Parameters

→ *`fsCreator`*
: Creator of the file system to call. A value of zero (0) tells the VFS Manager to check each registered file system, looking for one which supports the call.

→ *`apiCreator`*
: Registered creator ID.

→ *`apiSelector`*
Custom operation to perform.

↔ *`valueP`*
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.

↔ *`valueLenP`*
On entry, points to the size of the `valueP` buffer. On exit, this value reflects the size of the data written to the `valueP` buffer. If `valueLenP` is `NULL`, `valueP` is passed to the file system but is not updated on exit.

Returns Returns `errNone` if the operation completed successfully, or one of the following otherwise:

`expErrNotOpen`
The file system library necessary for this call has not been installed or has not been opened.

`expErrUnsupportedOperation`
The specified opcode and/or creator is unsupported or undefined.

`sysErrParamErr`
The *`valueP`* buffer is too small.

`vfsErrNoFileSystem`
VFS Manager cannot find an appropriate file system to handle the request.

Comments The driver identifies the call and its API by a registered creator ID and a selector. 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.

VFSDirCreate Function

Purpose Create a new directory.

Declared In `VFSMgr.h`

Prototype

```
status_t VFSDirCreate (uint16_t volRefNum,
    const char *dirNameP)
```

Parameters

→ *`volRefNum`*
Volume reference number returned from `VFSVolumeEnumerate()`.

→ *`dirNameP`*
Pointer to the full path of the directory to be created.

Returns Returns `errNone` if the operation completed successfully, or one of the following otherwise:

`expErrNotOpen`
The file system library necessary for this call has not been installed or has not been opened.

`vfsErrBadName`
Some or all of the path, up to but not including the last component specified in the *`dirNameP`* parameter, does not exist.

`vfsErrFileAlreadyExists`
A file with this name already exists in this location.

`vfsErrNoFileSystem`
The VFS Manager cannot find an appropriate file system to handle the request.

`vfsErrVolumeBadRef`
The volume has not been mounted.

`vfsErrVolumeFull`
There is not enough space left on the volume.

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()`.

VFSDirEntryEnumerate Function

Purpose Enumerate the entries in a given directory. Entries can include files, links, and other directories.

Declared In `VFSMgr.h`

Prototype

```
status_t VFSDirEntryEnumerate (FileRef dirRef,
    uint32_t *dirEntryIteratorP,
    FileInfoType *infoP)
```

Parameters

→ *`dirRef`*
: Directory reference returned from `VFSFileOpen()`.

↔ *`dirEntryIteratorP`*
: Pointer to the index of the last entry enumerated. For the first iteration, initialize this parameter to the constant `vfsIteratorStart`. Upon return, this references the next entry in the directory. If `infoP` is the last entry, this parameter is set to `vfsIteratorStop`.

↔ *`infoP`*
: Pointer to the `FileInfoType` data structure that contains information about the given directory entry. The `nameP` and `nameBufLen` fields in this structure must be initialized prior to calling `VFSDirEntryEnumerate`.

Returns Returns `errNone` if the operation completed successfully, or one of the following otherwise:

`expErrEnumerationEmpty`
: There are no directory entries left to enumerate.

`expErrNotOpen`
: The file system library necessary for this call has not been installed or has not been opened.

`sysErrParamErr`
: The *`dirEntryIteratorP`* is not valid.

`vfsErrFileBadRef`
: The specified file reference is invalid.

`vfsErrIsNotADirectory`
: The specified file reference is valid, but does not point to a directory.

`vfsErrNoFileSystem`
The VFS Manager cannot find an appropriate file system to handle the request.

Comments The directory to be enumerated must first be opened with `VFSFileOpen()` in order to obtain a file reference. In order to obtain 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 `vfsIteratorStart` and `vfsIteratorStop`. Before the first call to `VFSDirEntryEnumerate`, `dirEntryIteratorP` should be initialized to `vfsIteratorStart`. Each iteration then changes the value pointed to by `dirEntryIteratorP`. When information on the last entry in the directory is returned, `dirEntryIteratorP` is set to `vfsIteratorStop`.

WARNING! 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 code excerpt illustrates how to use `VFSDirEntryEnumerate`.

```
FileInfoType info;
FileRef dirRef;
UInt32 dirIterator;
char *fileName = MemPtrNew(256);  // should check for err

// open the directory first, to get the directory reference
// volRefNum must have already been defined
err = VFSFileOpen(volRefNum, "/", vfsModeRead, &dirRef);
if(err == errNone) {

   info.nameP = fileName;    // point to local buffer
   info.nameBufLen = 256;
   dirIterator = vfsIteratorStart
   while (dirIterator != vfsIteratorStop) {
      // Get the next file
      err = VFSDirEntryEnumerate (dirRef, &dirIterator,
         &info);
      if (err == errNone) {
         // Do something with the directory entry information
         // Pull the attributes from info.attributes
         // The file name is in fileName
      } else {
         // handle error, possibly by breaking out of the
loop
      }
   } else {
      // handle directory open error here
   }
   MemPtrFree(fileName);
}
```

VFSExportDatabaseToFile Function

Purpose Save the specified database to a PDB or PRC file on an external storage card.

Declared In `VFSMgr.h`

Prototype `status_t VFSExportDatabaseToFile (uint16_t` *`volRefNum`*`, const char *`*`pathNameP`*`, DatabaseID` *`dbID`*`)`

Parameters

→ *`volRefNum`*
Volume on which the destination file should be created.

→ *`pathNameP`*
Pointer to the complete path and name of the destination file to be created.

→ *`dbID`*
ID of the database being exported.

Returns Returns `errNone` if the operation completed successfully, or one of the following otherwise:

`expErrNotEnoughPower`
There is insufficient battery power to perform the database export operation.

`vfsErrBadName`
The path name specified in *`pathNameP`* is not valid.

Comments This utility function exports a database from main memory to a PDB or PRC file on an external storage card. This function is the opposite of `VFSImportDatabaseFromFile()`. It first creates the file specified in the `pathNameP` parameter with `VFSFileCreate()`.

After opening the file the Exchange Manager function `ExgDBWrite()` is called with an internal callback function for exporting the file from the Data Manager. The Exchange Manager makes repeated calls to this callback function, which receives the data back in blocks. Once all the data has been exported, VFS Manager closes the file.

This function is used, for example, to copy applications from main memory to a storage card.

See Also `VFSExportDatabaseToFileCustom()`, `VFSFileWrite()`, `VFSImportDatabaseFromFile()`

VFSExportDatabaseToFileCustom Function

Purpose Save the specified database to a PDB or PRC file on an external storage card. This function differs from `VFSExportDatabaseToFile()` in that it allows you to track the progress of the export operation.

Declared In `VFSMgr.h`

Prototype

```
status_t VFSExportDatabaseToFileCustom
   (uint16_t volRefNum, const char *pathNameP,
   DatabaseID dbID, VFSExportProcPtr exportProcP,
   void *userDataP)
```

Parameters

→ *volRefNum*
Volume on which the destination file should be created.

→ *pathNameP*
Pointer to the complete path and name of the destination file to be created.

→ *dbID*
ID of the database being exported.

→ *exportProcP*
User-defined callback function that tracks the progress of the export. This function should allow the user to cancel the export. Pass `NULL` if you don't have a progress callback function. See `VFSExportProcPtr()` for the requirements of this function.

→ *`userDataP`*
Pointer to any data you want to pass to the callback function specified in `exportProcP`. This information is not used internally by the VFS Manager. Pass `NULL` if you don't have a progress callback function or if that function doesn't need any such data.

Returns Returns `errNone` if the operation completed successfully, or one of the following otherwise:

`expErrNotEnoughPower`
There is insufficient battery power to perform the database export operation.

`vfsErrBadName`
The path name specified in *`pathNameP`* is not valid.

This function can also return any error code other than `errNone` produced by your callback function.

Comments This function is similar to `VFSExportDatabaseToFile()` in that it exports a database from main memory to a PDB or PRC file on an external storage card. It extends the functionality by allowing you to specify a callback function that tracks the progress of the export. It first creates the file specified in the `pathNameP` parameter with `VFSFileCreate()`. After opening the file, the Exchange Manager function `ExgDBWrite()` is called with an internal callback function for exporting the file from the Data Manager. Exchange Manager makes repeated calls to this function, which receives the data back in blocks. The progress tracker, if one has been specified, is also called every time a new chunk of data is passed back. Once all the data has been exported, the VFS Manager closes the file.

See Also `VFSExportDatabaseToFile()`, `VFSFileWrite()`, `VFSImportDatabaseFromFileCustom()`

VFSExportDatabaseToFileCustomV40 Function

Purpose Save the specified database to a PDB or PRC file on an external storage card. This function differs from

`VFSExportDatabaseToFile()` in that it allows you to track the progress of the export operation.

Declared In `VFSMgr.h`

Prototype

```
status_t VFSExportDatabaseToFileCustomV40
    (uint16_t volRefNum, const char *pathNameP,
    uint16_t cardNo, LocalID dbID,
    VFSExportProcPtr exportProcP, void *userDataP)
```

Parameters

→ *`volRefNum`*
Volume on which the destination file should be created.

→ *`pathNameP`*
Pointer to the complete path and name of the destination file to be created.

→ *`cardNo`*
Card number on which the PDB or PRC being exported resides.

→ *`dbID`*
ID of the database being exported.

→ *`exportProcP`*
User-defined callback function that tracks the progress of the export. This function should allow the user to cancel the export. Pass `NULL` if you don't have a progress callback function. See `VFSExportProcPtr()` for the requirements of this function.

→ *`userDataP`*
Pointer to any data you want to pass to the callback function specified in `exportProcP`. This information is not used internally by the VFS Manager. Pass `NULL` if you don't have a progress callback function or if that function doesn't need any such data.

Returns Returns `errNone` if the operation completed successfully, or one of the following otherwise:

`expErrNotEnoughPower`
There is insufficient battery power to perform the database export operation.

`vfsErrBadName`
The path name specified in *`pathNameP`* is not valid.

This function can also return any error code other than `errNone` produced by your callback function.

Comments This function is similar to `VFSExportDatabaseToFile()` in that it exports a database from main memory to a PDB or PRC file on an external storage card. It extends the functionality by allowing you to specify a callback function that tracks the progress of the export. It first creates the file specified in the `pathNameP` parameter with `VFSFileCreate()`. After opening the file, the Exchange Manager function `ExgDBWrite()` is called with an internal callback function for exporting the file from the Data Manager. Exchange Manager makes repeated calls to this function, which receives the data back in blocks. The progress tracker, if one has been specified, is also called every time a new chunk of data is passed back. Once all the data has been exported, the VFS Manager closes the file.

This function is used, for example, to copy applications from main memory to a storage card.

Compatibility This function is only provided for compatibility with previous versions of Palm OS; the *`cardNo`* parameter is ignored.

See Also `VFSExportDatabaseToFile()`, `VFSFileWrite()`, `VFSImportDatabaseFromFileCustom()`

VFSExportDatabaseToFileV40 Function

Purpose Save the specified database to a PDB or PRC file on an external storage card.

Declared In `VFSMgr.h`

Prototype

```
status_t VFSExportDatabaseToFileV40
   (uint16_t volRefNum, const char *pathNameP,
   uint16_t cardNo, LocalID dbID)
```

Parameters

→ *`volRefNum`*
Volume on which the destination file should be created.

→ *`pathNameP`*
Pointer to the complete path and name of the destination file to be created.

→ *`cardNo`*
Card number on which the PDB or PRC being exported resides.

→ *`dbID`*
ID of the database being exported.

Returns Returns `errNone` if the operation completed successfully, or one of the following otherwise:

`expErrNotEnoughPower`
There is insufficient battery power to perform the database export operation.

`vfsErrBadName`
The path name specified in *`pathNameP`* is not valid.

Comments This utility function exports a database from main memory to a PDB or PRC file on an external storage card. This function is the opposite of `VFSImportDatabaseFromFile()`. It first creates the file specified in the `pathNameP` parameter with `VFSFileCreate()`. After opening the file the Exchange Manager function `ExgDBWrite()` is called with an internal callback function for exporting the file from the Data Manager. The Exchange Manager makes repeated calls to this callback function, which receives the data back in blocks. Once all the data has been exported, VFS Manager closes the file.

This function is used, for example, to copy applications from main memory to a storage card.

Compatibility This function is only provided for compatibility with previous versions of Palm OS; the *`cardNo`* parameter is ignored.

See Also `VFSExportDatabaseToFileCustom()`, `VFSFileWrite()`, `VFSImportDatabaseFromFile()`

VFSFileClose Function

Purpose Close a file or directory that has been opened with `VFSFileOpen()`.

Declared In `VFSMgr.h`

Prototype `status_t VFSFileClose (FileRef fileRef)`

Parameters → *`fileRef`*
File reference number returned from `VFSFileOpen()`.

Returns Returns `errNone` if the operation completed successfully, or one of the following otherwise:

`expErrNotOpen`
The file system library necessary for this call has not been installed or has not been opened.

`vfsErrFileBadRef`
The specified file reference is invalid.

VFSFileCreate Function

Purpose Create a file. This function cannot be used to create a directory; use `VFSDirCreate()` instead.

Declared In `VFSMgr.h`

Prototype
```
status_t VFSFileCreate (uint16_t volRefNum,
   const char *pathNameP)
```

Parameters → *`volRefNum`*
Reference number of the volume on which to create the file. This volume reference number is returned from `VFSVolumeEnumerate()`.

→ *`pathNameP`*
Pointer to the full path of the file to be created. All parts of the path, excluding the filename, must already exist.

Returns Returns `errNone` if the operation completed successfully, or one of the following otherwise:

`expErrNotOpen`
The file system library necessary for this call has not been installed or has not been opened.

`vfsErrBadName`
  The *`pathNameP`* is invalid.

`vfsErrFileAlreadyExists`
  A file with this name already exists in this location.

`vfsErrNoFileSystem`
  The VFS Manager cannot find an appropriate file system to handle the request.

`vfsErrVolumeBadRef`
  The volume has not been mounted.

`vfsErrVolumeFull`
  There is not enough space left on the volume.

Comments It is the responsibility of the file system library 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 "Naming Files" on page 80 for a description of how to construct a valid path.

This function does not open the file. Use `VFSFileOpen()` to open the file.

This function should not be used to create a directory. To create a directory use `VFSDirCreate()`.

See Also `VFSFileDelete()`

VFSFileDBGetRecord Function

Purpose Load a record from an opened PDB file on an external card into the storage heap.

Declared In `VFSMgr.h`

Prototype

```
status_t VFSFileDBGetRecord (FileRef ref,
   uint16_t recIndex, MemHandle *recHP,
   uint8_t *recAttrP, uint32_t *uniqueIDP)
```

Parameters → *`ref`*
  The file reference returned from `VFSFileOpen()`. Note that the open file must be a PDB file.

→ *`recIndex`*
The index of the record to load.

← *`recHP`*
Pointer to the record data's handle in the storage heap. If `NULL` is returned in this parameter there is either no data in this field or an error occurred reading this data from the file. If the handle is not `NULL`, you must dispose of the allocated handle using `MemHandleFree()`.

← *`recAttrP`*
Pointer to the attributes of the record. The values returned are identical to the atttributes returned from `DmRecordInfoV50()`. See "Non-Schema Database Record Attributes" on page 108 for a description of each attribute. Pass `NULL` for this parameter if you do not want to retrieve this information.

← *`uniqueIDP`*
Pointer to the unique identifier for this record. Pass `NULL` for this parameter if you do not want to retrieve this information.

Returns Returns `errNone` if the operation completed successfully, or one of the following otherwise:

`dmErrIndexOutOfRange`
The *`recIndex`* is out of range.

`dmErrNotRecordDB`
The file referenced by *`ref`* is not a record database.

`memErrNotEnoughSpace`
There is not enough space in memory for the requested record entry.

`sysErrParamErr`
A `NULL` value was passed in for the *`recHP`*, *`recAttrP`*, and *`uniqueIDP`* parameters.

`vfsErrBadData`
The local offsets (`localChunkID`) from the top of the PDB to the start of the raw record data for this entry are out of order.

Comments This function is analogous to `DmGetRecord()` but works with files on an external card rather than databases in main memory. This function allocates a handle of the appropriate size from the storage heap and returns it in the `recHP` parameter. The caller is responsible for freeing this memory, using `MemHandleFree()`, when it is no longer needed.

NOTE: This function is not efficient for multiple accesses and should be used sparingly.

See Also `VFSFileReadData()`

VFSFileDBGetResource Function

Purpose Load a resource into the storage heap from an opened PRC file.

Declared In `VFSMgr.h`

Prototype

```
status_t VFSFileDBGetResource (FileRef ref,
   DmResourceType type, DmResourceID resID,
   MemHandle *resHP)
```

Parameters

→ *ref*
The file reference returned from `VFSFileOpen()`. Note that the open file must be a `PRC` file.

→ *type*
The type of resource to load. See Chapter 2, "Palm OS Databases," for more information on resources.

→ *resID*
The ID of resource to load.

← *resHP*
Pointer to the resource data handle that was loaded into memory.

Returns Returns `errNone` if the operation completed successfully, or one of the following otherwise:

`dmErrNotResourceDB`
The file referenced by *ref* is not a resource database.

`dmErrResourceNotFound`
The requested resource was not found.

`memErrNotEnoughSpace`
There is not enough space in memory for the requested resource entries.

`sysErrParamErr`
`resHP` is `NULL`.

Comments This function locates the specified resource in the open PRC file. See *Exploring Palm OS: Palm OS File Formats* for more information on the layout of PRC files.

Once the resource is found, `VFSFileDBGetResource` allocates a handle of the appropriate size in the storage heap and reads it into memory. The handle to this memory location is returned through the `resHP` parameter. The caller is responsible for freeing this memory, using `MemHandleFree()`, when it is no longer needed.

NOTE: This function is not efficient for multiple accesses and should be used sparingly.

VFSFileDBInfo Function

Purpose Get information about a database represented by an open PRC or PDB file.

Declared In `VFSMgr.h`

Prototype

```
status_t VFSFileDBInfo (FileRef ref, char *nameP,
   uint16_t *attributesP, uint16_t *versionP,
   uint32_t *crDateP, uint32_t *modDateP,
   uint32_t *bckUpDateP, uint32_t *modNumP,
   MemHandle *appInfoHP, MemHandle *sortInfoHP,
   uint32_t *typeP, uint32_t *creatorP,
   uint16_t *numRecordsP)
```

Parameters → *`ref`*
The file reference returned from `VFSFileOpen()`. Note that the open file must be a PRC or PDB file.

← `nameP`
Pointer to a 32-byte character array in which the database name is returned. Pass `NULL` for this parameter if you do not want to retrieve the database name.

← `attributesP`
Pointer to the database attributes stored in the file. The values returned are identical to the atttributes returned from `DmDatabaseInfo()`. See "Database Attributes" on page 109 for a description of each attribute. Pass `NULL` for this parameter if you do not want to retrieve the database's attributes.

← `versionP`
Pointer to the application-specific version number of the database. The default version number is zero (0). Pass `NULL` for this parameter if you do not want to retrieve the version number.

← `crDateP`
Pointer to the date the database was created, expressed in seconds since midnight (00:00:00) January 1, 1904. Pass `NULL` for this parameter if you do not want to retrieve the creation date.

← `modDateP`
Pointer to the date the database was last modified, expressed in seconds since midnight (00:00:00) January 1, 1904. A database's modification date is updated only if a change has been made to the database when it is opened with write access. Pass `NULL` for this parameter if you do not want to retrieve the database's modification date.

← `bckUpDateP`
Pointer to the date the database was last backed up, expressed in seconds since midnight (00:00:00) January 1, 1904. Pass `NULL` for this parameter if you do not want to retrieve the database's backup date.

← `modNumP`
Pointer to the number of times the database was modified. This number is updated every time a record is added, modified, or deleted. Pass `NULL` for this parameter if you do not want to retrieve the modification count.

← *appInfoHP*
Pointer to the application info block handle. If `NULL` is returned in this parameter, either there is no data in this field or an error occurred reading this data from the file. If a value other than `NULL` is returned, you must dispose of the allocated handle using `MemHandleFree()`. If you do not want to retrieve the application info block, pass `NULL` for this parameter.

← *sortInfoHP*
Pointer to the sort info block handle. If `NULL` is returned in this parameter, either there is no data in this field or an error occurred reading this data from the file. If a value other than `NULL` is returned, you must dispose of the allocated handle using `MemHandleFree()`. Pass `NULL` for this parameter if you do not want to retrieve the sort info block handle.

← *typeP*
Pointer to the type of database as it was created. This may be a user-defined database type or a database type defined by Palm OS. Some of the more common database types returned here are:

`'appl'`	Standard Palm™ application (resource database)
`'libr'`	Standard shared library
`'libf'`	File system shared library
`'libs'`	Slot driver shared library
`'data'`	Standard Palm data file (record database)

Pass `NULL` for this parameter if you do not want to retrieve the database's type.

← *creatorP*
Pointer to the database's creator. Pass NULL for this parameter if you do not want to retrieve this information.

← *`numRecordsP`*
Pointer to the number of records in the database. Pass `NULL` for this parameter if you do not want to retrieve this information.

Returns Returns `errNone` if the operation completed successfully, or one of the following otherwise:

`memErrNotEnoughSpace`
There is not enough space in memory for the database header.

`vfsErrBadData`
The file referenced by the *`ref`* parameter is too small to contain a database header, or the database header is corrupted.

Comments This function is analogous to `DmDatabaseInfo()`, but it works with files on an external card rather than with databases in main memory. See *Exploring Palm OS: Palm OS File Formats* for a description of the header block in PRC and PDB files.

See Also `VFSFileGetAttributes()`, `VFSFileGetDate()`

VFSFileDelete Function

Purpose Delete a closed file or directory.

Declared In `VFSMgr.h`

Prototype
```
status_t VFSFileDelete (uint16_t volRefNum,
    const char *pathNameP)
```

Parameters
→ *`volRefNum`*
Volume reference number returned from `VFSVolumeEnumerate()`.

→ *`pathNameP`*
Pointer to the full path of the file or directory to be deleted.

Returns Returns `errNone` if the operation completed successfully, or one of the following otherwise:

`expErrNotOpen`
The file system library necessary for this call has not been installed or has not been opened.

`vfsErrBadName`
: The path name specified in `pathNameP` is not valid.

`vfsErrDirNotEmpty`
: The directory being deleted is not empty.

`vfsErrFileStillOpen`
: The file is still open.

`vfsErrFileNotFound`
: The file could not be found.

`vfsErrFilePermissionDenied`
: The requested permissions could not be granted.

`vfsErrNoFileSystem`
: The VFS Manager cannot find an appropriate file system to handle the request.

`vfsErrVolumeBadRef`
: The volume has not been mounted.

VFSFileEOF Function

Purpose Get end-of-file status for an open file. This function only operates on files and cannot be used with directories.

Declared In `VFSMgr.h`

Prototype `status_t VFSFileEOF (FileRef` *`fileRef`*`)`

Parameters → *`fileRef`*
: File reference returned from `VFSFileOpen()`.

Returns Returns `errNone` if the operation completed successfully, or one of the following otherwise:

`vfsErrFileEOF`
: The file pointer is at the end of file.

`expErrNotOpen`
: The file system library necessary for this call has not been installed or has not been opened.

`vfsErrFileBadRef`
: The specified file reference is invalid.

`vfsErrIsADirectory`
: The specified file reference points to a directory instead of a file. This is an invalid operation on a directory.

`vfsErrNoFileSystem`
: The VFS Manager cannot find an appropriate file system to handle the request.

VFSFileGetAttributes Function

Purpose Obtain the attributes of an open file or directory.

Declared In `VFSMgr.h`

Prototype

```
status_t VFSFileGetAttributes (FileRef fileRef,
   uint32_t *attributesP)
```

Parameters

→ *`fileRef`*
: File reference returned from `VFSFileOpen()`.

← *`attributesP`*
: Pointer to the attributes associated with the file or directory. See "File and Directory Attributes" on page 413 for a list of values that can be returned through this parameter.

Returns Returns `errNone` if the operation completed successfully, or one of the following otherwise:

`expErrNotOpen`
: The file system library necessary for this call has not been installed or has not been opened.

`vfsErrFileBadRef`
: The specified file reference is invalid.

`vfsErrNoFileSystem`
: The VFS Manager cannot find an appropriate file system to handle the request.

See Also VFSFileDBInfo(), VFSFileGetDate(), VFSFileSetAttributes()

VFSFileGetDate Function

Purpose Obtain the dates on an open file or directory.

Declared In `VFSMgr.h`

Prototype
```
status_t VFSFileGetDate (FileRef fileRef,
   uint16_t whichDate, uint32_t *dateP)
```

Parameters

→ *`fileRef`*
File reference returned from `VFSFileOpen()`.

→ *`whichDate`*
Specifies which date—creation, modification, or last access—you want. Supply one of the values listed under "Date Types" on page 414.

← *`dateP`*
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.

Returns Returns `errNone` if the operation completed successfully, or one of the following otherwise:

`expErrNotOpen`
The file system library necessary for this call has not been installed or has not been opened.

`expErrUnsupportedOperation`
The specified date type is not supported by the underlying file system.

`vfsErrFileBadRef`
The specified file reference is invalid.

`sysErrParamErr`
The *`whichDate`* parameter is not one of the defined constants.

Comments 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`.

See Also VFSFileDBInfo(), VFSFileGetAttributes(), VFSFileSetDate()

VFSFileOpen Function

Purpose Open a file or directory and returns a reference for it.

Declared In `VFSMgr.h`

Prototype

```
status_t VFSFileOpen (uint16_t volRefNum,
   const char *pathNameP, uint16_t openMode,
   FileRef *fileRefP)
```

Parameters

→ *`volRefNum`*
The volume reference number returned from `VFSVolumeEnumerate()`.

→ *`pathNameP`*
Pointer to the full path of the file or directory to be opened. This must be a valid path. It cannot be empty and can not contain null characters. The format of the pathname should match what the underlying file system supports. See "Naming Files" on page 80 for a description of how to construct a valid path.

→ *`openMode`*
Mode to use when opening the file. See "Open Mode Constants" on page 412 for a list of accepted modes.

← *`fileRefP`*
Pointer to the opened file or directory reference which is supplied to various other `VFSFile...` operations. This value is filled in on return.

Returns Returns `errNone` if the operation completed successfully, or one of the following otherwise:

`expErrCardReadOnly`
The open mode requested includes write access but the file is read-only.

`expErrNotOpen`
The file system library necessary for this call has not been installed or has not been opened.

`vfsErrBadName`
The *`pathNameP`* parameter is invalid.

`vfsErrFileNotFound`
The specified file or directory could not be found.

`vfsErrFilePermissionDenied`
: The file cannot be opened in the requested open mode, or it has already been opened with `vfsModeExclusive`.

`vfsErrVolumeBadRef`
: The specified volume has not been mounted.

See Also `VFSFileClose()`, `VFSDirEntryEnumerate()`, `VFSFileOpenFromURL()`

VFSFileOpenFromURL Function

Purpose Open a file or directory given a URL to that file or directory.

Declared In `VFSMgr.h`

Prototype
```
status_t VFSFileOpenFromURL
    (const char *fileURLP, uint16_t openMode,
    FileRef *fileRefP, uint16_t *numOccurrencesP)
```

Parameters

→ *`*fileURLP`*
: URL to the file or directory to be opened. This must be a valid URL. It cannot be empty and can not contain null characters.

→ *`openMode`*
: Mode to use when opening the file. See "Open Mode Constants" on page 412 for a list of accepted modes.

← *`fileRefP`*
: Pointer to the opened file or directory reference number which can then be supplied to various other `VFSFile...` operations. This value is filled in on return.

← *`numOccurrencesP`*
: The number of files the URL matched. Set this pointer to `NULL` if you don't need this information.

Returns Returns `errNone` if the operation completed successfully, or one of the following otherwise:

`expErrCardReadOnly`
: The open mode requested includes write access but the file is read-only.

`expErrNotOpen`
: The file system library necessary for this call has not been installed or has not been opened.

`vfsErrBadName`
: The *`pathNameP`* parameter is invalid.

`vfsErrFileNotFound`
: The specified file or directory could not be found.

`vfsErrFilePermissionDenied`
: The file cannot be opened in the requested open mode, or it has already been opened with `vfsModeExclusive`.

`vfsErrVolumeBadRef`
: The specified volume has not been mounted.

Comments `VFSOpenFileFromURL()` exists to aid a higher-level entity, such as the Exchange Manager, in opening a file referenced in a URL such as `file:///VolumeName/PALM/Launcher/myApp.prc` (see *Exploring Palm OS: High-Level Communications* for a specification of the URL format) This function differs from `VFSFileOpen()` in its use of a volume name (in the URL) instead of a volume reference number to differentiate the card. This difference allows the URL to be saved in a "bookmark" and later re-used to open the same file; this wouldn't work with volume reference numbers since they change with every insertion and removal of a card. In the case where multiple cards with the same volume name are present in a device at the same time, each card is checked for the presence of the file, and if multiple instances of the same file are found on these different cards the one with the most recent modification date is opened and returned. In this instance the optional *`numOccurrencesP`* parameter is set to the number of matching files found.

See Also `VFSFileClose()`, `VFSDirEntryEnumerate()`, `VFSFileOpen()`

VFSFileRead Function

Purpose Read data from a file into the dynamic heap. This function only operates on files and cannot be used with directories; use `VFSDirEntryEnumerate()` to explore the contents of a directory.

Declared In `VFSMgr.h`

Prototype

```
status_t VFSFileRead (FileRef fileRef,
   uint32_t numBytes, void *bufP,
   uint32_t *numBytesReadP)
```

Parameters

→ *`fileRef`*
  File reference returned from `VFSFileOpen()`.

→ *`numBytes`*
  Number of bytes to read.

← *`bufP`*
  Pointer to the destination chunk where the data is to be stored. This can be a pointer to any writable memory.

← *`numBytesReadP`*
  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. Pass `NULL` for this parameter if you do not need to know how many bytes were read.

Returns Returns `errNone` if the operation completed successfully, or one of the following otherwise:

`expErrNotOpen`
  The file system library necessary for this call has not been installed or has not been opened.

`vfsErrFileBadRef`
  The specified file reference is invalid.

`vfsErrFileEOF`
  The end of the file has been reached.

`vfsErrFilePermissionDenied`
  Read permission is not enabled for this file.

`vfsErrIsADirectory`
  The specified file reference is for a directory instead of a file. This is an invalid operation on a directory.

`vfsErrNoFileSystem`
The VFS Manager cannot find an appropriate file system to handle the request.

Comments The file system does not use `DmWrite()` and cannot be used to read data into the storage heap.

See Also VFSFileReadData(), VFSFileWrite(), VFSImportDatabaseFromFile()

VFSFileReadData Function

Purpose Read data from a file into a chunk of memory in the storage heap. This function only operates on files and cannot be used with directories; use `VFSDirEntryEnumerate()` to explore the contents of a directory.

Declared In `VFSMgr.h`

Prototype

```
status_t VFSFileReadData (FileRef fileRef,
   uint32_t numBytes, void *bufBaseP,
   uint32_t offset, uint32_t *numBytesReadP)
```

Parameters

→ *`fileRef`*
File reference returned in `VFSFileOpen()`.

→ *`numBytes`*
Number of bytes to read.

← *`bufBaseP`*
Pointer to the destination chunk in the storage heap where the data is to be stored. This pointer must be obtained through the appropriate call to the Memory Manager API.

→ *`offset`*
Offset, in bytes, within the *`bufBaseP`* chunk where the data is to be written.

← *`numBytesReadP`*
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. Pass `NULL` for this parameter if you do not need to know how many bytes were read.

Returns Returns `errNone` if the operation completed successfully, or one of the following otherwise:

`expErrNotOpen`
: The file system library necessary for this call has not been installed or has not been opened.

`vfsErrFileBadRef`
: The specified file reference is invalid.

`vfsErrFileEOF`
: The end of the file has been reached.

`vfsErrFilePermissionDenied`
: Read permission is not enabled for this file.

`vfsErrIsADirectory`
: The specified file reference is for a directory instead of a file. This is an invalid operation on a directory.

`vfsErrNoFileSystem`
: The VFS Manager cannot find an appropriate file system to handle the request.

Comments When data is read from an external card with `VFSFileReadData`, it is copied into a chunk of memory in the storage heap. This chunk **must** be allocated by the application before the call to `VFSFileReadData()`. This function calls `DmWrite()` to put the data in the storage heap.

See Also VFSFileRead(), VFSFileWrite()

VFSFileRename Function

Purpose Rename a closed file or directory. This function cannot be used to move a file to another directory within the file system.

Declared In `VFSMgr.h`

Prototype
```
status_t VFSFileRename (uint16_t volRefNum,
   const char *pathNameP, const char *newNameP)
```

Parameters → *`volRefNum`*
: Volume reference number returned from `VFSVolumeEnumerate()`.

→ *`pathNameP`*
Pointer to the full path of the file or directory to be renamed.

→ *`newNameP`*
Pointer to the new filename. Note that this is the name of the file only and does not include the path to the file.

Returns Returns `errNone` if the operation completed successfully, or one of the following otherwise:

`expErrNotOpen`
The file system library necessary for this call has not been installed or has not been opened.

`vfsErrBadName`
The name provided in either *`pathNameP`* or *`newNameP`* is invalid. This is also returned if the string pointed to by *`newNameP`* is a path, rather than a filename.

`vfsErrFileAlreadyExists`
A file with the new name already exists in this location.

`vfsErrFileNotFound`
The source file could not be found.

`vfsErrFilePermissionDenied`
Write permission is not enabled for this file.

`vfsErrNoFileSystem`
The VFS Manager cannot find an appropriate file system to handle the request.

`vfsErrVolumeBadRef`
The volume has not been mounted.

`vfsErrVolumeFull`
There is not enough space left on the volume.

Comments **WARNING!** This function invalidates directory enumeration. You cannot continue enumerating files after renaming one of them with this function. If you need to operate on additional files in the directory, you must first restart the enumeration.

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 != errNone) {
      // handle error...
}
```

VFSFileResize Function

Purpose Change the size of an open file. This function only operates on files and cannot be used with directories.

Declared In `VFSMgr.h`

Prototype
```
status_t VFSFileResize (FileRef fileRef,
   uint32_t newSize)
```

Parameters

→ *`fileRef`*
File reference returned from `VFSFileOpen()`.

→ *`newSize`*
The desired new size of the file. This can be larger or smaller then the current file size.

Returns Returns `errNone` if the operation completed successfully, or one of the following otherwise:

`expErrNotOpen`
The file system library necessary for this call has not been installed or has not been opened.

`vfsErrFileBadRef`
The specified file reference is invalid.

`vfsErrIsADirectory`
The specified file reference points to a directory instead of a file. This is an invalid operation on a directory.

`vfsErrNoFileSystem`
The VFS Manager cannot find an appropriate file system to handle the request.

`vfsErrVolumeFull`
: There is not enough space left on the volume.

Comments The location of the file pointer is undefined after a call to this function.

See Also VFSFileSize()

VFSFileSeek Function

Purpose Set the position within an open file from which to read or write. This function only operates on files and cannot be used with directories.

Declared In `VFSMgr.h`

Prototype

```
status_t VFSFileSeek (FileRef fileRef,
   FileOrigin origin, int32_t offset)
```

Parameters

→ *`fileRef`*
: File reference returned from `VFSFileOpen()`.

→ *`origin`*
: Origin to use when calculating the new position. The `offset` parameter indicates the desired new position relative to this origin, which can be one of the constants listed under "Seek Origins" on page 415.

→ *`offset`*
: Offset, either positive or negative, from the origin to which the current position should be set. A value of zero (0) positions you at the specified origin.

Returns Returns `errNone` if the operation completed successfully, or one of the following otherwise:

`expErrNotOpen`
: The file system library necessary for this call has not been installed or has not been opened.

`vfsErrFileBadRef`
: The specified file reference is invalid.

`vfsErrFileEOF`
: The file pointer is at the end of file.

`vfsErrIsADirectory`
: The specified file reference points to a directory instead of a file. This is an invalid operation on a directory.

`sysErrParamErr`
: The specified origin is not one of the defined constants.

Comments During a call to this function, if the resulting position would be beyond the end of the file, it sets the position to the end of the file.

See Also VFSFileSize(), VFSFileTell()

VFSFileSetAttributes Function

Purpose Change the attributes of an open file or directory.

Declared In `VFSMgr.h`

Prototype

```
status_t VFSFileSetAttributes (FileRef fileRef,
   uint32_t attributes)
```

Parameters

→ *`fileRef`*
: File reference returned from `VFSFileOpen()`.

→ *`attributes`*
: Attributes to associate with the file or directory. See "File and Directory Attributes" on page 413 for a list of values you can use when setting this parameter:

Returns Returns `errNone` if the operation completed successfully, or one of the following otherwise:

`expErrNotOpen`
: The file system library necessary for this call has not been installed or has not been opened.

`sysErrParamErr`
: One of the parameters is invalid.

`vfsErrFileBadRef`
: The specified file reference is invalid.

`vfsErrNoFileSystem`
: `The VFS Manager cannot find an appropriate file system to handle the request.`

Comments

NOTE: 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()`. This function may fail when setting other attributes, depending on the underlying file system.

See Also VFSFileGetAttributes(), VFSFileSetDate()

VFSFileSetDate Function

Purpose Change the dates on an open file or directory.

Declared In `VFSMgr.h`

Prototype

```
status_t VFSFileSetDate (FileRef fileRef,
   uint16_t whichDate, uint32_t date)
```

Parameters

→ *`fileRef`*
File reference returned in `VFSFileOpen()`.

→ *`whichDate`*
Specifies which date—creation, modification, or last access—to modify. Supply one of the values listed under "Date Types" on page 414.

→ *`date`*
The new date. This field should be expressed in the standard Palm OS date format — the number of seconds since midnight (00:00:00) January 1, 1904.

Returns Returns `errNone` if the operation completed successfully, or one of the following otherwise:

`expErrNotOpen`
The file system library necessary for this call has not been installed or has not been opened.

`expErrUnsupportedOperation`
The specified date type is not supported by the underlying file system.

`sysErrParamErr`
: The *`whichDate`* parameter is not one of the defined constants.

`vfsErrFileBadRef`
: The specified file reference is invalid.

`vfsErrFilePermissionDenied`
: Write permission is not enabled for this file.

`vfsErrNoFileSystem`
: The VFS Manager cannot find an appropriate file system to handle the request.

Comments 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`.

See Also VFSFileGetDate(), VFSFileSetAttributes()

VFSFileSize Function

Purpose Obtain the size of an open file. This function only operates on files and cannot be used with directories.

Declared In `VFSMgr.h`

Prototype

```
status_t VFSFileSize (FileRef fileRef,
   uint32_t *fileSizeP)
```

Parameters

→ *`fileRef`*
: File reference returned from `VFSFileOpen()`.

← *`fileSizeP`*
: Pointer to the size of the open file.

Returns Returns `errNone` if the operation completed successfully, or one of the following otherwise:

`expErrNotOpen`
: The file system library necessary for this call has not been installed or has not been opened.

`vfsErrFileBadRef`
: The specified file reference is invalid.

`vfsErrIsADirectory`
: The specified file reference points to a directory instead of a file. This is an invalid operation on a directory.

`vfsErrNoFileSystem`
: The VFS Manager cannot find an appropriate file system to handle the request.

See Also VFSFileResize(), VFSFileTell(), VFSVolumeSize()

VFSFileTell Function

Purpose Get the current position of the file pointer within an open file. This function only operates on files and cannot be used with directories.

Declared In `VFSMgr.h`

Prototype

```
status_t VFSFileTell (FileRef fileRef,
   uint32_t *filePosP)
```

Parameters

→ *`fileRef`*
: File reference returned from `VFSFileOpen()`.

← *`filePosP`*
: Pointer to the current file position.

Returns Returns `errNone` if the operation completed successfully, or one of the following otherwise:

`expErrNotOpen`
: The file system library necessary for this call has not been installed or has not been opened.

`vfsErrFileBadRef`
: The specified file reference is invalid.

`vfsErrIsADirectory`
: The specified file reference points to a directory instead of a file. This is an invalid operation on a directory.

`vfsErrNoFileSystem`
: The VFS Manager cannot find an appropriate file system to handle the request.

See Also VFSFileSeek(), VFSFileSize()

VFSFileWrite Function

Purpose Write data to an open file. This function only operates on files and cannot be used with directories.

Declared In `VFSMgr.h`

Prototype

```
status_t VFSFileWrite (FileRef fileRef,
   uint32_t numBytes, const void *dataP,
   uint32_t *numBytesWrittenP)
```

Parameters

→ *`fileRef`*
File reference returned from `VFSFileOpen()`.

→ *`numBytes`*
The number of bytes to write.

→ *`dataP`*
Pointer to the data that is to be written.

← *`numBytesWrittenP`*
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. Pass `NULL` for this parameter if you do not need to know how many bytes were written.

Returns Returns `errNone` if the operation completed successfully, or one of the following otherwise:

`expErrNotOpen`
The file system library necessary for this call has not been installed or has not been opened.

`vfsErrFileBadRef`
The specified file reference is invalid.

`vfsErrFilePermissionDenied`
Write permission is not enabled for this file.

`vfsErrIsADirectory`
The specified file reference points to a directory instead of a file. This is an invalid operation on a directory.

`vfsErrNoFileSystem`
The VFS Manager cannot find an appropriate file system to handle the request.

`vfsErrVolumeFull`
: There is not enough space left on the volume.

See Also VFSExportDatabaseToFile(), VFSExportDatabaseToFileCustom(), VFSFileRead(), VFSFileReadData()

VFSGetDefaultDirectory Function

Purpose Determine the default location on the given volume for files of a particular type.

Declared In `VFSMgr.h`

Prototype
```
status_t VFSGetDefaultDirectory
    (uint16_t volRefNum, const char *fileTypeStr,
    char *pathStr, uint16_t *bufSizeP)
```

Parameters

→ *`volRefNum`*
: Volume reference number returned from `VFSVolumeEnumerate()`.

→ *`fileTypeStr`*
: 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`.”

← *`pathStr`*
: Pointer to the buffer which receives the default directory path for the requested file type.

← *`bufSizeP`*
: Pointer to the size of the path (including the null terminator). Set this to the size of *`pathStr`* buffer on input. Reflects the number of bytes copied to *`pathStr`* on output. Note that if truncation occurred the actual length of the string might be less than indicated by this value.

Returns Returns `errNone` if the operation completed successfully, or one of the following otherwise:

`vfsErrBadName`
: There is no default directory registered for the requested file type.

`vfsErrBufferOverflow`
A match was found, but the *`pathStr`* buffer is too small to hold the resulting path string. A partial path is returned in *`pathStr`*.

`vfsErrFileNotFound`
No match was found for the specified volume. The error could have occurred with either the media type specified for this volume or the file type requested.

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.

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.

See Also VFSDirEntryEnumerate(), VFSRegisterDefaultDirectory(), VFSUnregisterDefaultDirectory()

VFSImportDatabaseFromFile Function

Purpose Create a database from a PDB or PRC file on an external storage card.

Declared In `VFSMgr.h`

Prototype
```
status_t VFSImportDatabaseFromFile
   (uint16_t volRefNum, const char *pathNameP,
   DatabaseID *dbIDP)
```

Parameters → *`volRefNum`*
Volume on which the source file resides.

→ *`pathNameP`*
Pointer to the full path and name of the source file.

← *`dbIDP`*
: 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 returned along with the error `dmErrAlreadyExists`.

Returns Returns `errNone` if the operation completed successfully, or one of the following otherwise:

`dmErrAlreadyExists`
: The PRC or PDB file already exists in the storage heap. In this case *`dbIDP`* is set to point to the existing file.

`expErrNotEnoughPower`
: There is insufficient battery power to complete the requested operation.

`vfsErrBadName`
: The path name specified in *`pathNameP`* is not valid.

Comments This utility function imports a PDB or PRC file resident on an external storage card into a new database in the storage heap. It first calls `VFSFileOpen()` to open the file specified in `pathNameP`. Assuming that a corresponding PRC or PDB does not already exist in the storage heap, `VFSImportDatabaseFromFile()` calls the Exchange Manager function `ExgDBRead()` with an internal callback function for importing a file to the Data Manager. The Exchange Manager makes repeated calls to this function, which passes the data back in blocks. Once the file has been successfully imported, the owner (the imported file, if it's an executable, or the associated application if it is not) is sent a `sysAppLaunchCmdSyncNotify` launch code to make it aware of the new database.

This function only imports the specified PDB or PRC file; it does not import bundled databases or overlays. If there are bundled databases and/or overlays associated with the PDB or PRC file you are importing, you will need to write additional code to explicitly handle them.

This function doesn't provide any progress indication to the user. If you need to provide feedback to the user as the file import progresses, use `VFSImportDatabaseFromFileCustom()` instead.

This function is used, for example, to copy applications from a storage card to main memory.

See Also VFSExportDatabaseToFile(), VFSFileRead()

VFSImportDatabaseFromFileCustom Function

Purpose Create a database from the specified PDB or PRC file on an external storage card. This function differs from `VFSImportDatabaseFromFile()` in that it allows you to track the progress of the import operation.

Declared In `VFSMgr.h`

Prototype

```
status_t VFSImportDatabaseFromFileCustom
   (uint16_t volRefNum, const char *pathNameP,
   DatabaseID *dbIDP,
   VFSImportProcPtr importProcP, void *userDataP)
```

Parameters

→ *`volRefNum`*
Volume on which the source file resides.

→ *`pathNameP`*
Pointer to the full path and name of the source file.

← *`dbIDP`*
Pointer to the 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 returned along with the error `dmErrAlreadyExists`.

→ *`importProcP`*
User-defined callback function that tracks the progress of the import. This function should allow the user to cancel the import. Pass `NULL` if you don't have a progress callback function. See `VFSImportProcPtr()` for the requirements of this function.

→ *`userDataP`*
Pointer to any data you want to pass to the callback function specified in *`importProcP`*. This information is not used internally by the VFS Manager. Pass `NULL` if you don't have a progress callback function, or if that function doesn't need any such data.

Returns Returns `errNone` if the operation completed successfully, or one of the following otherwise:

`vfsErrBadName`
: The path name specified in *`pathNameP`* is not valid.

`expErrNotEnoughPower`
: The power required to import a database is not available.

`dmErrAlreadyExists`
: The PRC or PDB file already exists in main memory. In this case the *`cardNoP`* and *`dbIDP`* are set to point to the existing file.

Comments This function is similar to `VFSImportDatabaseFromFile()` in that it imports a PDB or PRC file on an external storage card into a new database on the storage heap. It extends the functionality by allowing you to specify a callback function that tracks the progress of the export. It first calls `VFSFileOpen()` to open the file specified in *`pathNameP`*. If a corresponding PRC or PDB does not already exist in main memory, it calls the Exchange Manager function `ExgDBRead()` with an internal callback function for importing the file from the Data Manager. The Exchange Manager makes repeated calls to this function, which receives the data back in blocks. The progress tracker, if one has been specified, is also called every time a new chunk of data is passed back. Once the file has been successfully imported, the owner (the imported file, if it's an executable, or the associated application if it is not) is sent a `sysAppLaunchCmdSyncNotify` launch code to make it aware of the new database.

Like `VFSImportDatabaseFromFile`, this function only imports the specified PDB or PRC file; it does not import bundled databases or overlays.

This function is used, for example, to copy applications from a storage card to main memory.

See Also VFSFileRead(), VFSExportDatabaseToFileCustom()

VFSImportDatabaseFromFileCustomV40 Function

Purpose Create a database from the specified PDB or PRC file on an external storage card. This function differs from `VFSImportDatabaseFromFile()` in that it allows you to track the progress of the import operation.

Declared In `VFSMgr.h`

Prototype

```
status_t VFSImportDatabaseFromFileCustomV40
   (uint16_t volRefNum, const char *pathNameP,
   uint16_t *cardNoP, LocalID *dbIDP,
   VFSImportProcPtr importProcP, void *userDataP)
```

Parameters

→ *`volRefNum`*
Volume on which the source file resides.

→ *`pathNameP`*
Pointer to the full path and name of the source file.

← *`cardNoP`*
Pointer to the variable that receives the card number of the newly-created database. If the database already resides in the storage heap, the card number of the existing database is returned along with the error `dmErrAlreadyExists`.

← *`dbIDP`*
Pointer to the 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 returned along with the error `dmErrAlreadyExists`.

→ *`importProcP`*
User-defined callback function that tracks the progress of the import. This function should allow the user to cancel the import. Pass `NULL` if you don't have a progress callback function. See `VFSImportProcPtr()` for the requirements of this function.

→ *`userDataP`*
Pointer to any data you want to pass to the callback function specified in *`importProcP`*. This information is not used internally by the VFS Manager. Pass `NULL` if you don't have a progress callback function, or if that function doesn't need any such data.

Returns Returns `errNone` if the operation completed successfully, or one of the following otherwise:

`vfsErrBadName`
: The path name specified in *`pathNameP`* is not valid.

`expErrNotEnoughPower`
: The power required to import a database is not available.

`dmErrAlreadyExists`
: The PRC or PDB file already exists in main memory. In this case the *`cardNoP`* and *`dbIDP`* are set to point to the existing file.

Comments This function is similar to `VFSImportDatabaseFromFile()` in that it imports a PDB or PRC file on an external storage card into a new database on the storage heap. It extends the functionality by allowing you to specify a callback function that tracks the progress of the export. It first calls `VFSFileOpen()` to open the file specified in *`pathNameP`*. If a corresponding PRC or PDB does not already exist in main memory, it calls the Exchange Manager function `ExgDBRead()` with an internal callback function for importing the file from the Data Manager. The Exchange Manager makes repeated calls to this function, which receives the data back in blocks. The progress tracker, if one has been specified, is also called every time a new chunk of data is passed back. Once the file has been successfully imported, the owner (the imported file, if it's an executable, or the associated application if it is not) is sent a `sysAppLaunchCmdSyncNotify` launch code to make it aware of the new database.

Like `VFSImportDatabaseFromFile`, this function only imports the specified PDB or PRC file; it does not import bundled databases or overlays.

This function is used, for example, to copy applications from a storage card to main memory.

Compatibility This function is only provided for compatibility with previous versions of Palm OS. The returned *`*cardNoP`* is always 0.

See Also VFSFileRead(), VFSExportDatabaseToFileCustom(), VFSImportDatabaseFromFileCustom()

VFSImportDatabaseFromFileV40 Function

Purpose Create a database from a PDB or PRC file on an external storage card.

Declared In `VFSMgr.h`

Prototype
```
status_t VFSImportDatabaseFromFileV40
    (uint16_t volRefNum, const char *pathNameP,
    uint16_t *cardNoP, LocalID *dbIDP)
```

Parameters

→ *`volRefNum`*
Volume on which the source file resides.

→ *`pathNameP`*
Pointer to the full path and name of the source file.

← *`cardNoP`*
Pointer to a variable that receives the card number of the newly-created database. If the database already resides in the storage heap, the card number of the existing database is returned along with the error `dmErrAlreadyExists`.

← *`dbIDP`*
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 returned along with the error `dmErrAlreadyExists`.

Returns Returns `errNone` if the operation completed successfully, or one of the following otherwise:

`dmErrAlreadyExists`
The PRC or PDB file already exists in the storage heap. In this case *`dbIDP`* is set to point to the existing file.

`expErrNotEnoughPower`
There is insufficient battery power to complete the requested operation.

`vfsErrBadName`
The path name specified in *`pathNameP`* is not valid.

Comments This utility function imports a PDB or PRC file resident on an external storage card into a new database in the storage heap. It first calls `VFSFileOpen()` to open the file specified in `pathNameP`. Assuming that a corresponding PRC or PDB does not already exist in the storage heap, `VFSImportDatabaseFromFile()` calls the

Exchange Manager function `ExgDBRead()` with an internal callback function for importing a file to the Data Manager. The Exchange Manager makes repeated calls to this function, which passes the data back in blocks. Once the file has been successfully imported, the owner (the imported file, if it's an executable, or the associated application if it is not) is sent a `sysAppLaunchCmdSyncNotify` launch code to make it aware of the new database.

This function only imports the specified PDB or PRC file; it does not import bundled databases or overlays. If there are bundled databases and/or overlays associated with the PDB or PRC file you are importing, you will need to write additional code to explicitly handle them.

This function doesn't provide any progress indication to the user. If you need to provide feedback to the user as the file import progresses, use `VFSImportDatabaseFromFileCustom()` instead.

This function is used, for example, to copy applications from a storage card to main memory.

Compatibility This function is only provided for compatibility with previous versions of Palm OS. The returned `*cardNoP` is always 0.

See Also VFSExportDatabaseToFile(), VFSFileRead(), VFSImportDatabaseFromFile()

VFSRegisterDefaultDirectory Function

Purpose Register a specific directory as the default location for files of a given type on a particular kind of external storage card. This function is generally called by a slot driver for files and media types that are supported by that slot driver.

Declared In `VFSMgr.h`

Prototype

```
status_t VFSRegisterDefaultDirectory
   (const char *fileTypeStr, uint32_t mediaType,
   const char *pathStr)
```

Parameters

→ *`fileTypeStr`*
Pointer to the file type to register. This is a null-terminated string that can either be a MIME media type/subtype pair, such as "image/jpeg", "text/plain", or "audio/basic"; or a file extension, such as "`.jpeg`".

→ *`mediaType`*
Type of card media for which the default directory is being associated. See "Defined Media Types" on page 262 in *Exploring Palm OS: System Management* for the list of accepted values.

→ *`pathStr`*
Pointer to the default directory path to be associated with the specified file type. This string must be null-terminated, and must be the full path to the directory.

Returns Returns `errNone` if the operation completed successfully, or one of the following otherwise:

`sysErrParamErr`
Either the *`fileTypeStr`* parameter is `NULL` or the *`pathStr`* parameter is `NULL`.

`vfsErrFileAlreadyExists`
A default directory has already been registered for this file type on the specified card media type.

Comments This function first verifies that a default directory has not already been registered for the specified combination of file type and media type, and returns `vfsErrFileAlreadyExists` if one has been registered. To change an existing entry in the registry, you must first remove the existing entry with a call to `VFSUnregisterDefaultDirectory()` before re-registering it with `VFSRegisterDefaultDirectory`.

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

NOTE: Registering a directory as the default location for files of a given type on a particular type of media doesn't automatically register that file type with HotSync Exchange. See "HotSync Exchange" on page 138 of *Exploring Palm OS: High-Level Communications* for information on registering file types with HotSync Exchange.

See Also VFSGetDefaultDirectory()

VFSUnregisterDefaultDirectory Function

Purpose Sever the association between a particular file type and a default directory for a given type of card media.

Declared In `VFSMgr.h`

Prototype

```
status_t VFSUnregisterDefaultDirectory
    (const char *fileTypeStr, uint32_t mediaType)
```

Parameters

→ *`fileTypeStr`*
Pointer to the file type with which the default directory is associated. This is a null-terminated string that can either be a MIME media type/subtype pair, such as "image/jpeg", "text/plain", or "audio/basic"; or a file extension, such as "`.jpeg`".

→ *`mediaType`*
Type of card media for which the default directory is associated. See "Defined Media Types" on page 262 in *Exploring Palm OS: System Management* for the list of accepted values.

Returns Returns `errNone` if the operation completed successfully, or one of the following otherwise:

`sysErrParamErr`
The *`fileTypeStr`* parameter is `NULL`.

`vfsErrFileNotFound`
: A default directory could not be found in the registry for the specified file and media type.

Comments **NOTE:** Caution is advised when using this function, since you may remove another application's registration, causing data to mysteriously disappear from those applications.

See Also VFSGetDefaultDirectory(), VFSRegisterDefaultDirectory()

VFSVolumeEnumerate Function

Purpose Enumerate the mounted volumes.

Declared In `VFSMgr.h`

Prototype
```
status_t VFSVolumeEnumerate
   (uint16_t *volRefNumP, uint32_t *volIteratorP)
```

Parameters

← *`volRefNumP`*
: Pointer to the reference number for the volume represented by the current enumeration, or `vfsInvalidVolRef` if there are no more volumes to be enumerated or an error occurred.

↔ *`volIteratorP`*
: Pointer to a variable that holds the index of the current enumeration. Set the variable to `vfsIteratorStart` prior to the first iteration. Each call to `VFSVolumeEnumerate` updates the variable to the index of the next volume. When the last volume is reached, the variable pointed to by `volIteratorP` is set to `vfsIteratorStop`.

Returns Returns `errNone` if the operation completed successfully, or one of the following otherwise:

`expErrEnumerationEmpty`
: There are no volumes to enumerate.

`sysErrParamErr`
: The value pointed to by *`volIteratorP`* is not valid. This error is also returned when *`volIteratorP`* is `vfsIteratorStop`.

Comments This function returns a pointer to the volume reference number in the *volRefNumP* parameter. In order to traverse all volumes you must make repeated calls to [VFSVolumeEnumerate()]() inside a loop. Before the first call to `VFSVolumeEnumerate`, the variable pointed to by *volIteratorP* should be initialized to `vfsIteratorStart`. Each iteration then increments *volIteratorP* to the next entry after updating *volRefNumP*. When the last volume is reached, `*`*volIteratorP* is set to `vfsIteratorStop`. If there are no volumes to enumerate, `VFSVolumeEnumerate` returns `expErrEnumerationEmpty` when first called.

Example Below is an example of how to use `VFSVolumeEnumerate`.

```
UInt16 volRefNum;
UInt32 volIterator = vfsIteratorStart;

while (volIterator != vfsIteratorStop) {
   err = VFSVolumeEnumerate(&volRefNum, &volIterator);
   if (err == errNone) {
      // Do something with the volRefNum
   } else {
      // handle error... possibly by
      // breaking out of the loop
   }
}
```

VFSVolumeFormat Function

Purpose Format and mount the volume installed in a given slot.

Declared In `VFSMgr.h`

Prototype
```
status_t VFSVolumeFormat (uint8_t flags,
    uint16_t fsLibRefNum,
    VFSAnyMountParamPtr vfsMountParamP)
```

Parameters

→ *flags*
Flags that control how the volume should be formatted. Currently, the only flag not reserved is `vfsMountFlagsUseThisFileSystem`. Pass this flag to cause the volume to be formatted using the file system specified by `fsLibRefNum`. Pass zero (0) to have the VFS

Manager attempt to format the volume using a file system appropriate to the slot.

→ *`fsLibRefNum`*
: Reference number of the file system library for which the volume should be formatted. If the `flags` field is not set to `vfsMountFlagsUseThisFileSystem`, this parameter is ignored.

↔ *`vfsMountParamP`*
: Parameters to be used when formatting the volume and when mounting the volume after it has been formatted. Supply a pointer to either a `VFSSlotMountParamType` or a `VFSPOSEMountParamType` structure. Note that you'll need to cast your structure pointer to a `VFSAnyMountParamPtr`. Set the `mountClass` field to the appropriate value: if you are mounting to an Expansion Manager slot, set `mountClass` to `VFSMountClass_SlotDriver` and initialize `slotLibRefNum` and `slotRefNum` to the appropriate values. See the descriptions of `VFSAnyMountParamType`, `VFSSlotMountParamType`, and `VFSPOSEMountParamType` for information on the fields that make up these data structures.

Returns Returns `errNone` if the operation completed successfully, or one of the following otherwise:

`expErrNotEnoughPower`
: There is insufficient battery power to format and/or mount a volume.

`expErrNotOpen`
: The file system library necessary for this call has not been installed or has not been opened.

`vfsErrNoFileSystem`
: The VFS Manager cannot find an appropriate file system to handle the request.

Comments The slot driver currently only supports one volume per slot. If the volume is successfully formatted and mounted, the reference number of the mounted volume is returned in `vfsMountParamP->volRefNum`. If the format is unsuccessful or cancelled, `vfsMountParamP->volRefNum` is set to `vfsInvalidVolRef`.

If `vfsMountFlagsUseThisFileSystem` is passed as a flag, `VFSVolumeFormat` attempts to format the volume using the file system library specified by *`fsLibRefNum`*. Typically the flag parameter is not set. In this case `VFSVolumeFormat` tries to find a compatible library to format the volume, as follows:

1. Check to see if the default file system library feature is set. If it is, and if that file system is installed, it is used to format the volume. You can set the default file system using `FtrSet()`; supply `sysFileCVFSMgr` for the feature creator, and `vfsFtrIDDefaultFS` for the feature number.
2. Check to see if any of the installed file systems are natively supported for the slot on which the VFS Manager is trying to format. If one of them is, it is used to format the volume.
3. If none of the installed file systems can perform the format using the slot's native type, a dialog displays warning the user that their media may become incompatible with other devices if they continue with the format. The user may continue or cancel the format. If the user chooses to continue, `VFSVolumeFormat` formats the volume using the first file system library that was installed.

When calling `VFSVolumeFormat`, the volume can either be mounted or unmounted. The underlying file system library call requires the volume to be unmounted. `VFSVolumeFormat` checks to see if the volume is currently mounted and unmounts it, if necessary, using `VFSVolumeUnmount()` before making the file system call. If the file system successfully formats the volume, `VFSVolumeFormat` mounts it and posts a `sysNotifyVolumeMountedEvent` notification.

Example The following code excerpt formats a volume on an Expansion Manager slot using a compatible file system.

```
VFSSlotMountParamType slotParam;
UInt32 slotIterator = expIteratorStart;

slotParam.vfsMountParamP.mountClass =
   VFSMountClass_SlotDriver;
err = ExpSlotEnumerate(&slotParam.slotRefNum,
   &slotIterator);
err = ExpSlotLibFind(slotParam.slotRefNum,
   &slotParam.slotLibRefNum);

err = VFSVolumeFormat(NULL, NULL,
   (VFSAnyMountParamPtr) & slotParam);
```

See Also VFSVolumeMount()

VFSVolumeGetLabel Function

Purpose Determine the volume label for a particular volume.

Declared In `VFSMgr.h`

Prototype `status_t VFSVolumeGetLabel (uint16_t` *`volRefNum`*`,`
`char *`*`labelP`*`, size_t` *`bufSize`*`)`

Parameters

→ *`volRefNum`*
Volume reference number returned from `VFSVolumeEnumerate()`.

← *`labelP`*
Pointer to a character buffer into which the volume name is placed.

→ *`bufSize`*
Length, in bytes, of the `labelP` buffer.

Returns Returns `errNone` if the operation completed successfully, or one of the following otherwise:

`expErrNotOpen`
The file system library necessary for this call has not been installed or has not been opened.

`vfsErrNoFileSystem`
The VFS Manager cannot find an appropriate file system to handle the request.

`vfsErrVolumeBadRef`
The specified volume has not been mounted.

`vfsErrBufferOverflow`
The value specified in *`bufSize`* is not big enough to receive the full volume label.

`vfsErrNameShortened`
There was an error reading the full volume name. A shortened version is being returned.

Comments Volume reference numbers can change each time you mount 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 lower case characters, in any character set as well as the following special characters: $ % ' - _ @ ~ ` ! ( ) ^ # & + , ; = [ ].

See Also VFSVolumeSetLabel()

VFSVolumeInfo Function

Purpose Get information about the specified volume.

Declared In `VFSMgr.h`

Prototype
```
status_t VFSVolumeInfo (uint16_t volRefNum,
   VolumeInfoType *volInfoP)
```

Parameters

→ *`volRefNum`*
Volume reference number returned from `VFSVolumeEnumerate()`.

← *`volInfoP`*
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.

Returns Returns `errNone` if the operation completed successfully, or one of the following otherwise:

`expErrNotOpen`
: The file system library necessary for this call has not been installed or has not been opened.

`vfsErrNoFileSystem`
: The VFS Manager cannot find an appropriate file system to handle the request.

`vfsErrVolumeBadRef`
: The specified volume reference number is invalid.

See Also VFSVolumeGetLabel(), VFSVolumeSize()

VFSVolumeMount Function

Purpose Mount the card's volume on the specified slot.

Declared In `VFSMgr.h`

Prototype

```
status_t VFSVolumeMount (uint8_t flags,
   uint16_t fsLibRefNum,
   VFSAnyMountParamPtr vfsMountParamP)
```

Parameters

→ *`flags`*
: Flags that control how the volume should be mounted. Currently, the only flag not reserved is `vfsMountFlagsUseThisFileSystem`. Pass this flag to cause the volume to be mounted using the file system specified by *`fsLibRefNum`*. Pass zero (0) to have the VFS Manager attempt to mount the volume using a file system appropriate for the slot.

→ *`fsLibRefNum`*
: Reference number of the file system library for which the volume should be mounted. If the `flags` field is not set to `vfsMountFlagsUseThisFileSystem`, this parameter is ignored.

↔ *`vfsMountParamP`*
: Parameters to be used when mounting the volume after it has been formatted. Supply a pointer to either a `VFSSlotMountParamType` or a `VFSPOSEMountParamType` structure. Note that you'll need to cast your structure pointer to a `VFSAnyMountParamPtr`. Set the `mountClass` field to the appropriate value: if you are

mounting to an Expansion Manager slot, set `mountClass` to `VFSMountClass_SlotDriver` and initialize `slotLibRefNum` and `slotRefNum` to the appropriate values. See the descriptions of `VFSAnyMountParamType`, `VFSSlotMountParamType`, and `VFSPOSEMountParamType` for information on the fields that make up these data structures.

Returns Returns `errNone` if the operation completed successfully, or one of the following otherwise:

`expErrNotEnoughPower`
There is insufficient battery power to mount a volume.

`expErrNotOpen`
The file system library necessary for this call has not been installed or has not been opened.

`sysErrParamErr`
`vfsMountParamP` was initialized to `NULL`.

`vfsErrNoFileSystem`
The VFS Manager cannot find an appropriate file system to handle the request.

`vfsErrVolumeStillMounted`
The volume is already mounted with a different file system than was specified in *`fsLibRefNum`*.

Comments The slot driver only supports one volume per slot. The reference number of the mounted volume is returned in `vfsMountParamP->volRefNum`. If `vfsMountFlagsUseThisFileSystem` is passed as a flag, `VFSVolumeMount` attempts to mount the volume using the file system library specified by `fsLibRefNum`. Otherwise `VFSVolumeMount` tries to find a file system library which is able to mount the volume. If none of the installed file system libraries is able to mount the volume, `VFSVolumeMount` attempts to re-format the volume (using `VFSVolumeFormat()`) and then mount it. If `VFSVolumeMount` manages to successfully mount the volume, it ends by posting a `sysNotifyVolumeMountedEvent` notification.

After VFSVolumeMount successfully mounts a volume, it broadcasts `sysNotifyVolumeMountedEvent`. The VFS Manager, upon being notified of this event, searches the newly-mounted volume for `/PALM/start.prc`. If `start.prc` is found in the `/`

PALM directory, the VFS Manager copies it to main memory and launches it. If `start.prc` is not found, the VFS Manager switches to the Launcher instead. This behavior can be overridden; see "Card Insertion and Removal" on page 61 of *Exploring Palm OS: System Management*.

When `VFSVolumeMount` is called, if the volume is already mounted with a different file system than was specified in `fsLibRefNum`, a `vfsErrVolumeStillMounted` error is returned. If the volume is already mounted with the same file system that is specified in `fsLibRefNum`, or if `vfsMountFlagsUseThisFileSystem` is not set, `VFSVolumeMount` returns `errNone` and sets `volRefNumP` to the reference number of the currently mounted volume.

Example The following code excerpt mounts a volume on an Expansion Manager slot using a compatible file system.

```
VFSSlotMountParamType slotParam ;
UInt32 slotIterator = expIteratorStart;

slotParam.vfsMountParamP.mountClass =
   VFSMountClass_SlotDriver;
err = ExpSlotEnumerate(&slotParam.slotRefNum,
   &slotIterator);
err = ExpSlotLibFind(slotParam.slotRefNum,
   &slotParam.slotLibRefNum);

err = VFSVolumeMount(NULL, NULL,
   (VFSAnyMountParamPtr) & slotParam);
```

See Also VFSVolumeFormat(), VFSVolumeUnmount()

VFSVolumeSetLabel Function

Purpose Change the volume label for a mounted volume.

Declared In `VFSMgr.h`

Prototype
```
status_t VFSVolumeSetLabel (uint16_t volRefNum,
   const char *labelP)
```

Parameters

→ *`volRefNum`*
Volume reference number returned from `VFSVolumeEnumerate()`.

→ *`labelP`*
Pointer to the label to be applied to the specified volume. This string must be null-terminated.

Returns Returns `errNone` if the operation completed successfully, or one of the following otherwise:

`expErrNotOpen`
The file system library necessary for this call has not been installed or has not been opened.

`vfsErrBadName`
The supplied label is invalid.

`vfsErrNameShortened`
Indicates that the label name was too long. A shortened version of the label name was used instead.

`vfsErrVolumeBadRef`
The specified volume has not been mounted.

Comments Volume labels can be up to 255 characters long. They can contain any normal character, including spaces and lower case characters, in any character set as well as the following special characters: $ % ' - _ @ ~ ` ! ( ) ^ # & + , ; = [ ]. See "Naming Volumes" on page 77 for guidelines on naming.

NOTE: Most clients 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()`.

See Also VFSVolumeGetLabel()

VFSVolumeSize Function

Purpose Determine the total amount of space on a volume, as well as the amount that is currently being used.

Declared In `VFSMgr.h`

Prototype
```
status_t VFSVolumeSize (uint16_t volRefNum,
   uint32_t *volumeUsedP, uint32_t *volumeTotalP)
```

Parameters

→ *`volRefNum`*
Volume reference number returned from `VFSVolumeEnumerate()`.

← *`volumeUsedP`*
Pointer to a variable that receives the amount of space, in bytes, in use on the volume.

← *`volumeTotalP`*
Pointer to a variable that receives the total amount of space on the volume, in bytes.

Returns Returns `errNone` if the operation completed successfully, or one of the following otherwise:

`expErrNotOpen`
The file system library necessary for this call has not been installed or has not been opened.

`vfsErrNoFileSystem`
The VFS Manager cannot find an appropriate file system to handle the request.

`vfsErrVolumeBadRef`
: The specified volume has not been mounted.

See Also `VFSVolumeInfo()`

VFSVolumeUnmount Function

Purpose Unmount the given volume.

Declared In `VFSMgr.h`

Prototype `status_t VFSVolumeUnmount (uint16_t` *`volRefNum`*`)`

Parameters → *`volRefNum`*
: Volume reference number returned from `VFSVolumeEnumerate()`.

Returns Returns `errNone` if the operation completed successfully, or one of the following otherwise:

`expErrNotOpen`
: The file system library necessary for this call has not been installed or has not been opened.

`vfsErrNoFileSystem`
: The VFS Manager cannot find an appropriate file system to handle the request.

`vfsErrVolumeBadRef`
: The specified volume has not been mounted.

Comments This function closes any opened files and posts a `sysNotifyVolumeUnmountedEvent` notification once the file system is successfully unmounted.

See Also VFSVolumeMount()

Application-Defined Functions

VFSExportProcPtr Function

Purpose User-defined callback function supplied to `VFSExportDatabaseToFileCustom()` that tracks the progress of the export.

Declared In `VFSMgr.h`

Prototype
```
status_t (*VFSExportProcPtr)
    (uint32_t totalBytes, uint32_t offset,
    void *userDataP)
```

Parameters

→ *`totalBytes`*
The total number of bytes being exported.

→ *`offset`*
Undefined.

→ *`userDataP`*
Pointer to any application-specific data passed to the callback function. This pointer may be `NULL` if your callback doesn't need any such data.

Returns Your progress tracker should allow the user to abort the export. Return `errNone` if the export should continue, or any other value to abort the export process. If you return a value other than `errNone`, that value will be returned by `VFSExportDatabaseToFileCustom()`.

Comments See "Progress Dialogs" on page 31 of *Exploring Palm OS: User Interface* for more information on writing a progress tracker.

See Also `VFSImportProcPtr()`

VFSImportProcPtr Function

Purpose User-defined callback function supplied to `VFSImportDatabaseFromFileCustom()` that tracks the progress of the import.

Declared In `VFSMgr.h`

Prototype

```
status_t (*VFSImportProcPtr)
    (uint32_t totalBytes, uint32_t offset,
    void *userDataP)
```

Parameters

→ *`totalBytes`*
The total number of bytes being imported.

→ *`offset`*
The number of bytes that have already been imported. This value, along with the total number of bytes being imported, allows you to inform the user how far along the import is.

→ *`userDataP`*
Pointer to NY application-specific data passed to the callback function. This pointer may be `NULL` if your callback doesn't need any such data.

Returns Your progress tracker should allow the user to abort the import. Return `errNone` if the import should continue, or any other value to abort the import process. If you return a value other than `errNone`, that value will be returned by `VFSImportDatabaseFromFileCustom()`.

Comments See "Progress Dialogs" on page 31 of *Exploring Palm OS: User Interface* for more information on writing a progress tracker.

See Also `VFSExportProcPtr()`

Index