Pilot Memory and Database Management

Kenneth Albanowski
kjahds@kjahds.com

The Pilot takes a very simple approach to dealing with data. It has no file systems, no flash RAM support, and no secondary storage. Instead, both transitory and long-term data is stored in memory that is preserved intact as long as the Pilot has power. Thus memory management, and its close relative database management are crucial topics for Pilot programmers.

The tools provided by the PalmOS to manipulate memory and databases are quite unlike the capabilities of environments such as NewtonScript, Perl, and even BASIC. In this article I give C programmers with basic Pilot experience a tutorial on using memory and databases and discuss the Pilot database functions. Since I cover so many functions, I often skip the precise API details and error values. You can easily look these up in Palm's documentation.

Chunks of Memory

Memory allocation on the Pilot comes in one flavor, the chunk, but chunks can be served up several different ways. A chunk is a block of memory up to 64 KB in size that can be movable or immovable, and allocated in either dynamic or storage memory. (Actually, chunks are part of a higher level of memory management called a heap, but heaps rarely impact the Pilot programmer. Palm's documentation has the details for the curious.)

Length

A chunk cannot be bigger than 64 KB, partly because many of the memory management structures use 16-bit integers. It isn't worth fretting about this limit since it isn't going to go away any time soon. There is no OS limit on the amount of available memory – the PalmOS can theoretically use up to 4 GB of memory, albeit in 64 KB chunks.

Movability

Chunks can be movable or immovable. The idea, familiar to MacOS programmers, is that the operating system moves chunks of memory around without complex memory management hardware (which the Pilot lacks). With movable chunks, the OS can deal more easily with low and badly-fragmented memory. However, moving chunks around invalidates any pointers to them, so the PalmOS uses handles, which are effectively pointers to chunk pointers. Each handle points to a fixed, very small piece of memory (actually part of the heap containing the chunk) that contains a pointer to the real chunk. With this scheme, the OS can move a chunk around and update only the second pointer to keep track of things.

Immovable chunks have none of these concerns. Fixed into one place in memory, they never go gallivanting off behind your back.

Location

The Pilot partitions available RAM into two sections: dynamic and storage memory. Dynamic memory is freely available for all programs to use, but is rather small (usually around 32 KB) and is automatically cleared when a program exits or the machine reboots. Except for its small size, this is similar to main memory on any desktop computer.

The rest of RAM is used for storage memory, which is limited to read-only access by special CPU circuitry. If you try to directly modify the contents of storage memory, the Pilot crashes with an unfriendly exception message. Storage memory is only cleared when you choose "wipe all memory" during a hard reset or remove all battery power.

Access to perfectly good RAM is limited because this is where all the important user data is kept. By making it difficult to erase or modify, the chance of a buggy program wiping out all of that data is quite low.

You can allocate chunks from storage memory as easily as from dynamic memory, and manipulate them with the same functions, unless you want to change the contents of a storage chunk. To do that, you need to use some special functions which I describe later.

Manipulating Immovable Memory

Let's start out with a simple task, allocating a 256-byte chunk of immovable memory:

```
VoidPtr p = MemPtrNew(256);
if (!p) {
  /* New failed */
}
```

This code should look familiar to any C programmer who has used the `malloc` function. We ask the OS for a chunk of memory, and it gives us a void pointer in return. `VoidPtr` is a shorthand name that the Pilot API uses for `void *`. There's no requirement you use Palm's types in your code, but since Palm uses them throughout their own code, it's probably worth getting into the habit for consistency. If the allocation fails, you get zero instead of a pointer, and your program has to deal with the lack of memory.

At some point after obtaining a chunk and using it, you need to return it to the OS:

```
MemPtrFree(p);
```

This is analogous to the `free` function in C. While `MemPtrFree` does return an error value, the only likely failure occurs if you pass an invalid pointer, one that was never an allocated block or has been already freed.

Similar to C's `realloc`, you can resize a previously allocated chunk of memory like this:

```
if (!MemPtrResize(p, 512)) {
  /* Resized failed */
}
```

As with `MemPtrNew`, you should always check the return value. Since immovable chunks can't float around, there's a good chance you won't be able to resize it. If the resizing fails, your chunk is still there, but with its original size and contents. Note that unlike the C `realloc` function, a new pointer is never returned and the original is always valid. You should try to resize a chunk, not allocate a new one and copy the data. The less memory your program uses, even temporarily, the better.

Manipulating Movable Memory

Movable chunks take a bit more work to use, but you should use them whenever possible – the OS copes better with low memory when it can move chunks around as needed.

To allocate a movable chunk, things start out in much the same fashion:

```
VoidHand h = MemHandleNew(384);
if (!h) {
  /* failed */
}
```

Here `VoidHand`, which you should always use when referring to handles, is not a simple `void *`, and should not be treated as such. You cannot use a handle directly as a `CharPtr`, or some other type. Instead, you need to lock the handle down, fastening it (temporarily) to one place in memory:

```
CharPtr c = MemHandleLock(h);
```

Once you do this, you can use the pointer to your heart's content, just like any immovable memory chunk (or a `malloc`'d block on another OS). However, remember that by locking the chunk down, you take away the important benefit of movable memory.

As soon as you're done using the contents of a locked chunk, unlock the handle:

```
MemHandleUnlock(h);
```

Alternatively, you can use the pointer to unlock the handle. This produces identical results, and might be more convenient in some circumstances:

```
MemPtrUnlock(c);
```

You can lock a chunk more then once in a row, but you must always unlock it exactly the same number of times as you lock it, lest it be forgotten in a locked or under-locked state. Leave a chunk locked for as short a time as reasonable. The contents of the chunk aren't changed by unlocking, since you are only telling the OS that you don't need it at this moment.

Resizing movable chunks is similar to resizing immovable chunks:

```
if (MemHandleResize(h, 492)) {
  /* failed, chunk left at original size */
}
```

However, with immovable chunks, if you first unlock the handle, there is a much greater chance of successfully resizing the chunk than for an immovable chunk. This is often the most important benefit of using movable chunks.

When you finish with a movable chunk and want it to go away, you free it as before, but with a different function:

```
MemHandleFree(h);
```

LocalIDs

Pointers and handles are all well and good, but there's a subtle problem with them: due to the way PalmOS arranges memory (specifically the way it sequentially assigns address space to each memory card), retaining memory addresses, in the form of handles or pointers in storage memory can be dangerous. If a Pilot device has two removable cards, you can exchange the cards, making stored addresses invalid.

Palm solved this problem by creating the `LocalID`, a special form of handle or pointer which is relative to the beginning of a card. Few functions accept `LocalID`s directly, but unlike pointers, they survive storage in removable cards unscathed, no matter where or how the cards are used. With `LocalID`s, the actual pointer is calculated from the `LocalID` and the starting address of the card to which it relates. To convert a chunk handle to a `LocalID` use `MemHandleToLocalID`. For a chunk pointer, use `MemPtrToLocalID`.

You have to use one of the `MemLocalIDTo...` functions to convert from a `LocalID` to a pointer or handle. The simplest is `MemLocalIDToGlobal`, which returns a pointer or handle, whichever was originally used. Alternatively, `MemLocalIDToLockedPtr` can be more convenient, since it always returns a pointer, locking a movable chunk if necessary, to provide the pointer. (Don't forget to unlock the chunk afterwards.) Note that all of these functions take both a `LocalID` and a card number as parameters.

I hear you asking: Why would I want to put pointers into storage memory? Maybe you don't, but the database manager does.

Database Overview

The Pilot, like most handheld computers, has a database management system to make it easy to store and retrieve information. The Pilot's databases are little more than tools to categorize and find chunks of memory. Unlike the Newton, Pilot databases have no fields, indexes, or other typical database attributes. Instead, they are only ordered arrays of chunk handles. You control any further subdivision or organization by using C structures. Despite these limitations, all Pilot data, including RAM and ROM applications, user data, and application preferences, is stored in databases.

Each database is itself a chunk that contains a header describing the database plus a list of the chunks that contain the database entries. A database header has eleven parts:

- Name
- Creator code
- Type code
- Creation, modification, and backup dates
- Version number
- Application information block
- Sort information block
- Attributes flag (describing several database options)
- Array of database entries

Name

The name of a database is a 31 character, `NULL`-terminated string in the same character set used by the rest of the Pilot. This string is used by many of the database functions to select and name databases. It may also be used by the Pilot's Memory application to show the name of a database, as well as by Palm's HotSync utility to name the database it is synchronizing.

Creator and Type

The creator and type codes are 32-bit integers which provide a method of grouping databases – all databases with the same creator code are treated as part of the same application. They also describe a database's purpose. For example, the OS uses the type code to determine if a resource database contains an application. Type and creator codes, although long integers, are usually referred to in C source code using wide character constants:

```
long Creator = 'YoYo'
long Typecode = 'appl';
```

Despite looking like strings, these statements are the exact equivalents of assigning the numeric constant `0x596F596F` to `Creator`, and `0x6170706C` to `Typecode`.

Some special type codes include "appl" for applications and "panl" for preference panels, a new type of application on PalmOS 2.0. The currently-assigned creator codes include "memo" for the MemoPad application and "exps" for the Expense application on new Pilots.

The card number, name, type, and creator codes have an important relationship: on every card, each database must have a distinct name (although the same name can occur on more then one card). However, more than one database on a card can have the same type and creator codes. Because it's important that the creator code and database names you use for your application do not conflict with that of any other applications, Palm has a registry at <http://www.usr.com/palm/crid.html>. I recommend you use it.

Dates

The creation, modification, and backup dates (stored as long integers containing the offset in seconds from 1/1/1904) are maintained by various portions of the OS. Unfortunately, as far as I know, the modification date is not usually updated. The backup date, which is updated by HotSync, is more likely to be accurate. Another subtle issue is that the time zones for these dates are unspecified, so they should be treated as wall time.

Version

The version number is a 16-bit unsigned integer that you may use as you wish, presumably to keep track of the version of a database. I don't believe the PalmOS uses this field for any purpose.

Application and Sort information

The application information block is a slot containing the LocalID of the handle of an optional storage chunk for a database. The purpose of this chunk is to retain data relevant to all entries in a database. For example, the built-in applications store their category names and view preferences in the application info block.

The sort information block is, in theory, designed to store a sorted array of record indexes so that you can quickly change their display order. In practice, none of the built-in applications use this block. Feel free to experiment with it.

Attributes

The attributes flag is a bit flag containing various database options. Though not usually manipulated by the programmer, it contains the following options:

dmHdrAttrResDB (0x0001)
Resource database (as opposed to record database).

`dmHdrAttrReadOnly (0x0002)`
Read only, so attempts to write to the database will fail.

`dmHdrAttrAppInfoDirty (0x0004)`
The application info block is dirty. This flag is used by applications and their synchronization conduits.

`dmHdrAttrBackup (0x0008)`
Automatically back up this database during synchronization.

`dmHdrAttrOKToInstallNewer (0x0010)`
A newer version of this database may be installed over an older one, even if the older one is open. This flag is only available on PalmOS 2.0 or higher.

`dmHdrAttrResetAfterInstall (0x0020)`
The Pilot should be reset after this database is installed. This flag is also only available on PalmOS 2.0 or higher.

`dmHdrAttrOpen (0x8000)`
The database is open.

The Entries

Finally, we get to the crucial element of any database: the contents. The PalmOS takes a somewhat unusual approach, using two separate database types. This has an impact on what the database entries look like:

- Resource databases are generally used to store read-only data (although there is nothing preventing you from modifying their contents) such as plug-in modules, bitmaps, and application code. Resources are referred to by index within the database or by type code and ID number. PRC files are stored in resource databases.
- Record databases are used to store appointments, memos, expenses, and other application data. Record databases have mechanisms to keep track of inserted, deleted, and secret entries, to record categories, and to assign a unique ID number to each record. For this reason, record databases are easily synchronizable, unlike resource databases. Records are referred to by index or by unique ID number. PDB files contain record databases.

Each type of database can store a total of 65,536 entries.

Resource Database Entries

Resource database entries are probably most familiar to Macintosh programmers, since they are similar to the resources in a Mac file's resource fork. Each entry has three parts, stored in the resource header:

- `Type`, a 32-bit integer,
- `ID`, a 16-bit integer, and
- `Data`, a handle to a movable chunk.

The `type` field is similar to the type and creator codes for a database, and should be manipulated in the same way, using wide character constants. `ID` is an arbitrary short integer which does not have to be sequential. The resource data is a handle to a chunk of storage memory somewhere on the same card as the database header card.

Record Database Entries

Record databases entries are more like records in a traditional database system, although they have features unique to the PalmOS. Each entry has three parts, stored in the record header:

- `Attribute`, an 8-bit integer,
- `UniqueID`, a 24-bit integer, and
- `Data`, a handle to a movable chunk.

`Attribute` is an eight-bit integer with two separate halves: the lower four bits contain a record's category number, while the upper four bits contain the record's attributes:

`dmRecAttrSecret (0x10)`
Record is secret.

`dmRecAttrBusy (0x20)`
Record is busy (locked).

`dmRecAttrDirty (0x40)`
Record has been modified and will be copied during the next synchronization.

`dmRecAttrDelete (0x80)`
Record has been deleted from the Pilot.

The secret bit is the only attribute normally touched by a programmer. Note that there is no separate attribute for archived records. A deleted record's chunk handle is normally removed from the record header. If a record is marked deleted and still has a chunk, it's treated as archived.

`UniqueID` is an identification number assigned to each record in a record database. The OS goes to some trouble to make sure each record within a database has a unique number. Once a record is deleted its ID is unlikely to be immediately reused. Due to the design of record entries, these IDs are only 24 bits long. `UniqueID`s are not kept unique across separate data bases, and `UniqueID` numbers probably start in the same place after a database is recreated (after a hard reset, for example). Since the PalmOS generates `UniqueID`s automatically, you don't need to worry about creating or maintaining them.

Creating, Opening, and Closing Databases

Let's create a simple database:

```
Err result = DmCreateDatabase(
  0, "MyDatabase", 'HHSy', 'MyDB', false);
```

This statement creates a database called "MyDatabase", with creator "HHSy" and type "MyDB", on card `0`. The fourth parameter set to false requests a record database (true means a resource database). As always, you should check the return value for an error condition. If there is no error, then you've created a database.

You don't have access to it until you open it:

```
LocalID id = DmFindDatabase(0, "MyDatabase");
DmOpenRef db = DmOpenDatabase(0, id, dmModeReadWrite);
```

The first call attempts to locate the database's own chunk and then pass it to `DmOpenDatabase` to actually open the database. `dmModeReadWrite` is self-explanatory; the other option is `dmModeReadOnly`. In both these calls, `0` is a card number, referring to the first (and presently only) card.

As usual, check the return value of each function. If they work properly, and your database exists, `db` now contains a `DmOpenRef`, a pointer you can use to refer to an open database.

When you are done with a database, close it:

```
DmCloseDatabase(db);
```

Once you close a database, the `DmOpenRef` becomes invalid, and you must go through the steps from the beginning if you want to reopen the database.

You can open a database in a few other ways. You can open the latest database with a particular type and creator on any card:

```
DmOpenRef db = DmOpenDatabaseByTypeCreator(
  'MyDB', 'HHSy', dmModeReadWrite);
```

If for some reason you want to step through the entire directory of databases on a card, without opening them (as before, `0` is a card number):

```
int Databases = DmNumDatabases(0);
int i;
for (i=0;i<Databases;i++)
  id = DmGetDatabase(0, i);
```

If you have a more specific goal in mind, like traversing all the databases belonging to a particular application, you can use `DmGetNextDatabaseByTypeCreator` to execute a search. Here's a quick example to find all databases with the "`HHSy`" creator code. The zero in this case is a wild-card saying any database type is acceptable:

```
Boolean newSearch = true;
DmSearchStateType search;
UInt card;
LocalID result;

while (DmGetNextDatabaseByTypeCreator(newSearch,
  &search, 0, 'HHSy', true, &card, &result)==0)
{  newSearch = false;
  /* found a database */
}
```

There's one last function I should mention: `DmCreateDatabaseFromImage`. This function takes a pointer to a block of memory and creates a database from the block's contents. The contents should be a verbatim PRC or PDB file, obtained using the various debugging and development tools, and inserted into your application as either a resource or constant string data. The standard applications use this technique to recreate their database when the Pilot is hard-reset.

Allocating and Using Storage Memory

All the memory allocation functions I describe in the first section of this article allocate from dynamic memory. As you recall, most of the Pilot's RAM is reserved for storage memory. All database entries, including the optional application and sort blocks, must be allocated from storage memory so they don't disappear when a program exits or the Pilot is rebooted. There are, however, some functions for allocating and managing storage chunks that are not part of a database.

(A note on error handling: some of the functions discussed below do not return an error code on failure, just zero. To find the exact reason why these functions fail, you need to call `DmGetLastErr`. Since this value is only set on failure, and not cleared on success, you should only rely on its value immediately after a call fails and returns zero.)

Here's how you allocate a movable chunk in storage memory apart from a database:

```
VoidHand h = DmNewHandle(db, 256);
if (!h) {
  /* Failed, DmGetLastErr() returns the error code */  }
```

The first parameter to `DmNewHandle` is the `DmOpenRef` of an open database. This is used only to ensure the chunk is allocated on the same card as the database (and therefore attachable to that database) and does not connect the new chunk to that database in any other way. You can manipulate this handle using exactly the same functions as any other handle, including freeing it with `MemHandleFree`.

Now that you have a storage chunk to play around with, you need to know how to write into it. You can read from storage memory without any special effort, but due to the write-protection in the CPU, you need to use a special set of three functions to write to the chunk:

```
VoidPtr p = MemHandleLock(h);
DmSet (p, 0, 2, 'X'); /* Similar to MemSet or memset */
DmWrite (p, 2, p, 2); /* Similar to MemMove or memcpy */
DmStrCopy (p, 4, "YZ"); /* Similar to StrCopy or strcpy */
```

Each of these functions takes a pointer to a locked movable chunk, and an offset within that chunk. You must put the offset in the second parameter, and don't specify any pointer arithmetic in the first parameter – `DmStrCopy(p+4, 0, "YZ")` is not valid. All three functions are carefully designed to make sure you can modify only the record you indicate, without damaging any other data. You can accidentally wipe out the contents of one record, but no others.

You must use these functions to write to a storage chunk. If you forget, the Pilot crashes, pointing out your misdeed. Unfortunately, Palm's Pilot Simulator, as well as the various emulators, are more forgiving and don't detect this error. Testing your software on the Pilot itself is very important.

Manipulating Database Information

Now let's look at the functions for manipulating the information in a database header. The most capable is `DmDatabaseInfo`. When given the `LocalID` for a database (such as `DmFindDatabase` returns) and the card number, this function lets you read all the information in the database header. To change this information, you need to use `DmSetDatabaseInfo`. `DmOpenDatabaseInfo` retrieves miscellaneous information about an open database, specifically the card it's on, its `LocalID`, what mode it's been opened with, and how many times it has been opened without closing.

Putting all of this together, here's how to add a new application info block to a previously opened database:

```
VoidHand h = DmNewHandle (db, 64);
LocalID ai = MemHandleToLocalID(h);
LocalID dbID;
UInt cardNo;
DmOpenDatabaseInfo (db, &dbID, NULL, NULL, &cardNo, NULL)
DmSetDatabaseInfo(cardNo, dbID, NULL, NULL, NULL, NULL,
  NULL, NULL, NULL, &ai, NULL, NULL, NULL);
```

This code allocates a new chunk of storage memory, converts the chunk's handle to a `LocalID`, gets the `LocalID` and card number of the open database, and stores the `LocalID` of the new chunk in the application-info slot of the database header.

After this operation, the chunk is automatically preserved and even backed up along with the rest of the database. The main thing to remember here is not to use `MemHandleFree`, since the database now owns the chunk, not you.

If the database has an application info block, then `DmGetAppInfoID` retrieves it. `DmDatabaseInfo` can do this as well, but `DmGetAppInfoID` is much simpler to use.

```
LocalID ai = DmGetAppInfoID(db);
if (ai) {
  /* do something to c */
  CharPtr c = MemLocalIDToLockedPtr(AI, 0);
  MemPtrUnlock(c);
}
```

If you want to remove the application-info block from a database, you have to use `DmSetDatabaseInfo`, except you pass `0` instead of a `LocalID`.

One last useful database information function is `DmDatabaseSize`, which tells you exactly how much storage memory a database is taking up.

Using Resources

Resources are the Pilot's workhorse for storing and retrieving large chunks of unchanging data, like executable programs, bitmaps, and so on, as well as information like application and system preferences.

Creation

Here's how to allocate a 128-byte resource database entry of type "Type" and ID 33 for database db:

```
VoidHand h = DmNewResource(db, 'Type', 33, 128);
if (!h) {
  /* creation failed, DmGetLastErr() contains error */
}
```

If you have a previously-allocated storage chunk, from DmNewHandle for example, you can accomplish the same thing by attaching it to the database with DmAttachResource. If you do that, however, remember that the database now owns the chunk, so don't try to free it.

Since we are working with resources and the resource functions, I should mention that all these functions fail if you use them on a record database. Always remember what type of database you are using.

Access

Once you identify some resources you want to retrieve and an open database containing them, the simplest method is DmGetResource:

```
VoidHand h = DmGetResource('Type', 33);
```

Notice that you don't need to give any database ID. The PalmOS automatically keeps track of all open resource databases and lets you easily search for a resource among all of them, starting with the most recently opened. If you want to search only the most recently opened database, use DmGet1Resource instead.

Since your application's code is stored in a database (which is, in turn, packed in a PRC file for distribution), and this database is automatically opened when your application is launched, resources compiled into your application are available using this method. The PalmOS uses this technique to fetch menu and form resources. You're welcome to compile custom resources into your applications for your own use.

If you want to find a particular type of resource, without knowing its ID number, DmFindResourceType lets you search for the first and subsequent resources of that type. This function, and DmFindResource, return an index, not a handle, into the resource database. To retrieve a resource by index, use DmGetResourceIndex. Use DmNumResources to step through the resources sequentially.

Once you have a resource handle, you can manipulate it like any other storage handle: lock it down, read it, and use DmWrite, DmSet, and DmStrCopy to change it. As soon as you are done reading or modifying the record (even temporarily), you should unlock it.

When you're finished working with a record, you must also release it. This additional step is required for resource database entries:

```
MemHandleUnlock(h);
DmReleaseResource(h);
```

When you release a handle, you must not continue to use it. If you need to access the record again, you have to start over with another call to DmGetResource or DmGetResourceIndex.

Deletion

If you want to get rid of a resource, you should call DmRemoveResource with a resource index. Alternately, you can detach the chunk from the database with DmDetachResource, leaving it to be deleted or attached to another database on the same card. (A chunk detached like this is equivalent to a chunk allocated with DmStorageNew.)

Resizing

You can resize a resource by using DmResizeResource:

```
int Index = DmFindResourceType(db, 'Type', 33);
h = DmResizeResource(db, Index, 65);
if (!h) {
  /* failed, the original record is unchanged*/
}
```

Don't try to resize a resource that is locked or not yet released, and always use the returned value. Don't use the original handle (if you have it) – the chunk may have been moved and resized. Consequently, the handle may have changed.

Modification

You can retrieve all the information on a resource (including type and ID), given an index and DmResourceInfo. Likewise, you can change any of this information with DmSetResourceInfo. Here's code that increments the ID of an existing resource (a rather odd thing to do, but a good example):

```
Int index = 2; /* third resource */
Int ID;
DmResourceInfo(db, index, NULL, &ID, NULL);
ID++;
DmSetResourceInfo(db, index, NULL, &ID);
```

Using Records

Resource databases, while uncomplicated and useful, are not sufficient for storing data that can be synchronized. Record databases are slightly more complicated to use, and have more details to keep track of, but provide the tools you need for synchronization.

Creation

Let's start from the beginning, once again, by creating a new 192-byte record at the end of an open record database:

```
UInt atIndex = 0xFFFF;
VoidHand h = DmNewRecord(db, &atIndex, 192);
```

The atIndex variable tells the function where you want to insert the record in the database. If the function succeeds, atIndex contains the actual index used. If you call DmNewRecord with atIndex positioned beyond the last record by using 0xFFFF, the function places the record at the very end of the database. Creating a new record at index 0 inserts the record at the very beginning, shifting all other records (if any) down one.

You can also attach a previously allocated storage chunk to the database using DmAttachRecord. You can use this function to insert a record, by passing NULL as the last parameter, or to replace an existing record by passing a pointer to a handle variable to receive the handle of the chunk detached from the database:

```
UInt at = 0;
VoidHand old;
if (DmAttachRecord(db, &at, h, &old)) {
  /* failed, chunk not attached */
}
```

Access

Once you put a record into a database, you may want to get it back out again. The simplest approach is to retrieve the record by index number (the first record in the database is index 0, the second is 1, and so on). You can find the total number of records with DmNumRecords):

```
VoidHand h = DmGetRecord(db, 1);
if (!h) {
  /* failed, no second record, or database not open */
}
```

You can also retrieve a record with DmFindRecordByID, which searches the database for a record with a particular ID and returns its index. This is often used by the Pilot's search mechanism – one part of your program searches for matching records, supplying the ID of those records, and another part displays matching records using the IDs.

Once you obtain a record, it is marked busy and cannot be fetched again until it is released. As soon as you're done with the record, you should unlock and release it:

```
MemHandleUnlock(h);
DmReleaseRecord(db, 1, false);
```

To release a record, you must pass to `DmReleaseRecord` the index of the record (and not its handle), and a `true` or `false` indicating whether the record was modified. If modified, the record's dirty attribute is set. This is the attribute that tells the synchronization system that a record has been changed.

If you do not want to modify a record, you can fetch it with `DmQueryRecord`, which returns the handle, but doesn't mark the record as busy, so you need not use `DmReleaseRecord`. If you do this, however, don't try to change the contents of the record. `DmQueryRecord` can retrieve a record even if it's already marked busy.

Most of the record functions take an index into the database, but on occasion you may only have a handle to a record chunk. You can retrieve the index from the handle with `DmSearchRecord`.

Deletion

When it's time to delete a record, you have many choices because of the synchronization tools and a user's ability to request that a record be archived instead of deleted outright.

The following three functions are called in an identical manner, with the `DmOpenRef` of an open database and the index of a record. (As usual when you are dealing with record or resource indexes, the record or resource should be unlocked and released.)

`DmRemoveRecord`
Delete the record completely, leaving no traces.

`DmDeleteRecord`
Delete the record contents, but leave the header around so a synchronization conduit can tell that the record was deleted. The record is moved to the end of the database.

`DmArchiveRecord`
Retain the record contents, but mark the record as archived and deleted. The record is moved to the end of the database.

A fourth function, `DmRemoveSecretRecords`, takes only a `DmOpenRef` and completely removes (in the same manner as `DmRemoveRecord`) all records marked as secret. (The Security application uses this function.)

If you've written a synchronization conduit, you should always use `DmDeleteRecord` (or `DmArchiveRecord` at the user's request), so that your conduit can easily keep track of records. Conversely, if you do not have a conduit, you should use `DmRemoveRecord`, since the other two functions leave traces of the record behind and your database accumulates deleted records if it is never synchronized.

You can also detach a record using `DmDetachRecord`, which has the same affect as `DmRemoveRecord`. As with `DmDetachResource`, you are free to reattach or delete this storage chunk once it is detached. If you merely want to move a record from one place to another within a database, `DmMoveRecord` lets you to do this.

Resizing

As with resource databases, you can resize records:

```
VoidHand h = DmResizeRecord(db, 1, 512);
if (!h) {
  /* resize failed */
}
```

You should check and use the handle returned, not any previous handle to that record. The act of resizing a record can move it to such a degree that the handle changes.

Modification

You can retrieve the attribute and unique ID of any record with `DmRecordInfo`, and modify this information with `DmSetRecordInfo`.

The most common use for these functions is to change the category or secrecy of a record:

```
UInt at = 3;
UByte attr;
DmRecordInfo (db, at, &attr, NULL, NULL);
attr &= ~dmRecAttrCategoryMask; /* remove category */
attr &= ~dmRecAttrSecret; /* remove secrecy */
attr |= currentCategory; /* and set category */
DmSetRecordInfo (db, at, &attr, NULL);
```

Sorting

The PalmOS provides two sort algorithms: a Quicksort and an insertion sort. `DmInsertionSort` maintains the order of records that compare the same, while `DmQuickSort` is generally faster, but randomizes such records. You call both in an identical manner. They use a comparison function to decide how to sort the records.

Here's an example that assumes each record starts with an integer. They are sorted in ascending order:

```
Int MyCompare(void * elem1, void * elem2, Int other)   {
 IntPtr e1 = elem1, e2 = elem2;
 if (*e1 < *e2)
  return -1;
 else if (*e1 > *e2)
  return 1;
 else
  return 0;
}
DmInsertionSort(db, MyCompare, 0);
```

The third parameter to your comparison function is simply copied directly from the third parameter to the sort function. It's for passing miscellaneous information to the comparison function. Your comparison function receives locked pointers to each pair of records, so you don't need to worry about locking or unlocking the records. Your function should test the records and return a number less than zero if the first sorts before the second, zero if they sort equally, and greater than zero if the first sorts after the second.

If you want to keep a database sorted, it may be quickest to insert a record in the right place to begin with, which you can do with `DmFindSortPosition`:

```
VoidHand h = DmNewStorage(db, 2);
UInt at = DmFindSortPosition(
 db, MemHandleLock(h), MyCompare, 0);
```

`DmFindSortPosition` uses your comparison function to find the right place in the database for your record. You may then insert the record with `DmAttachRecord`.

The sort functions, including `DmFindSortPosition`, have changed slightly in the 2.0 PalmOS, so if you are using the new SDK, be sure to check the documentation for these functions.

Categories

The PalmOS doesn't directly keep track of names assigned to a category. Instead, each record has a four-bit integer that is treated as the record category.

There are several functions that help you use this part of the category system. Given a record index and a category, `DmPositionInCategory` tells you how many records of that category precede the given one. For example, the 2.0 version of MemoPad numbers each memo like this:

```
char position[10];
index=DmPositionInCategory(db, CurrentRecord, category);
StrIToA (position, index+1);
```

Given an index, `DmQueryNextInCategory` returns the index of the next record (if any) within a particular category. `DmSeekRecordInCategory` works in a similar fashion, except that you can search backwards or forwards and skip over an initial set of records in the category. For example, if you want to skip over 10 records of a category (suppose that 10 items are displayed on the screen, and you want to find the next record), you can use:

```
UInt at = topRecord;
if (!DmSeekRecordInCategorydb, &at, 10,
  dmSeekForward, CurrentCategory)) {
  /* Success, at contains matching record */
}
```

`DmNumRecordsInCategory` counts the total number of records in a particular category, and `DmMoveCategory` changes all records with one category number to a different one.

If your program only needs to run under the PalmOS 2.0, you can use `DmDeleteCategory` to delete all the records in a particular category. If compatibility is an issue, you need to do this manually by looping over each record, making sure it isn't already deleted, checking its category with `DmRecordInfo`, and deleting it if it matches (the loop index is decremented after a deletion since all subsequent records are shifted up one when a record is deleted):

```
UInt index;
UInt maximum = DmNumRecords(db);
UByte attr;
for (index=0;index<maximum;index++) {
  if (DmRecordInfo(db, index, &attr, NULL, NULL)==0)
    if ((attr & dmRecAttrDelete) == 0)
      if ((attr & dmRecAttrCategoryMask) == category)
  DmDeleteRecord(db, index—);
}
```

Conclusion

I hope this article gives you information you can use to manage the Pilot's memory easily and efficiently. I've tried to touch briefly on all the relevant issues and functions, but you should also read Palm's documentation. If it's still not clear how any of these functions work, take a look at how Palm uses it in the sample code included in the SDK. ✔