

## UNIX and Pilots

Kevin L. Flynn  
Kodachi Systems Group  
flynn@kodachi.com

I don't have a Windows machine. Never have, probably never will. I have a SPARCstation and a Macintosh, and they generally do a much better job of meeting my needs than Windows ever has a chance of doing. The most notable exception, of late, is U.S. Robotics's Pilot.

When I saw my first Pilot, synchronization software was available only for (you guessed it) Windows. After agonizing for a while and checking out all the specifications I could find, I decided that the Pilot was a worthwhile machine, and that I would simply write my own software to let my UNIX machine talk to my Pilot.

This article is the first of a series talking about that software, and more generally about what's under the hood of HotSync, and what it takes to get some other systems talking to a Pilot. I take a fairly detailed look at some of the guts of PalmOS, and at some of the issues surrounding the design and implementation of the UNIX side. This first article covers some of the background issues.

Note that I'm not the only person working on a Pilot communications package for UNIX. As it happens, I'm not even the one with the most functional package right now. If you're simply looking to download software that can talk to your Pilot from UNIX, check out <ftp://ns1.pfnet.com/pub/PalmOS/> for the pilot-link software, maintained by Jeff Dionne and Kenneth Albanowski. If you're interested in learning about how it's done, though, read on.

### An Overview of HotSync

The main purpose of getting a desktop machine – UNIX or otherwise – to talk to the Pilot is so the desktop machine can exchange data with the Pilot. The major hook available for doing this is the HotSync mechanism built into the Pilot.

HotSync has three major components: its architecture, its protocols, and its data structures.

#### Architecture

It should come as no surprise that HotSync is designed with one machine acting as a client and the other machine acting as a server. (The client initiates an action by making requests, the server completes that action by responding to the requests.) However, it's interesting to note that the current implementation puts the server on the Pilot and the client on the desktop (see "Client or Server?" on the next page). This is oddly counter-intuitive in some respects, but it works.

Since the Pilot acts as the server during a HotSync, my UNIX application has to be constructed to act like a HotSync client.

#### Protocols

I spend quite a lot of time looking at the HotSync protocol stack later in this series of articles. For now, let's just take a brief look at DLP, the protocol at the top of the stack.

DLP stands for Desktop Link Protocol. It defines a set of requests, each with a specific meaning, and provides a way for the client to issue those requests to the server and get responses back. Requests can carry argument data to the server; likewise, responses can carry result data back to the client. All DLP operations consist of a request followed by a response. All DLP operations are initiated by the client.

The primitive operations going on during the Pilot synchronization process map very directly to DLP requests and responses (this is what it means to be at the top of the protocol stack). There are about 30 DLP requests, with corresponding responses, defined at present, though not all of these are used in every synchronization.

#### Data Structures

Synchronization mechanisms in general (including HotSync) operate by assuming that the client and the server agree on the broad structure

of the data they're handing back and forth. If they don't, either the client or the server (or both) need to convert the data into a format that the other understands.

Since the Pilot is much more limited in storage space and CPU cycles than the desktop, U.S. Robotics wisely chose to make the desktop convert to and from the Pilot's native formats. This makes the Pilot's life easier, but it makes our lives as HotSync client developers harder, since we need to understand the Pilot's internal structures.

The first big thing to bear in mind here is that the Pilot doesn't have disks, so it doesn't use a file paradigm (see "Disks or No Disks?" on page 35). PalmOS applications keep their persistent data in PalmOS databases. Since the databases live in RAM, the applications can and do modify them in place.

Conceptually, a PalmOS database consists of some data describing the database itself (the metadata) and a collection of records. Each record contains some data; exactly what the data is, and how it's formatted, is application-specific. For example, the Memo Pad creates a database where each record stores a single memo. Likewise, each To Do item gets a record in the To Do Items database.

So far, so good, but there's more. Remember, this is all stored in the Pilot's native format, and applications modify database records in place. This means that the bits in the records coming over the wire to the client are the raw binary representation of the data as they're actually stored in the Pilot's RAM. This means big-endian byte ordering and data aligned on 16-bit boundaries. My UNIX application has to be prepared to do byte-swapping and proper packing and unpacking if it's going to work.

One more important detail is that PalmOS organizes its memory according to the physical card that the memory is installed on. The Pilot has only one memory card, but PalmOS appears to be able to cope with up to 15 cards of up to 256 MB each. Cards are identified by integers beginning with zero (so only card #0 is present in the Pilot). Although the user never has to care about which card holds any given database, the HotSync process does.

### PalmOS Database Internals

Since PalmOS databases are at the core of the synchronization process, it's worth taking a more detailed look at them. Unfortunately, the conceptual model above is a gross oversimplification – the real world is rather more complex.

The biggest complicating factor is that PalmOS supports two different flavors of databases. One flavor is called a resource database; the other isn't consistently called anything, but it's a database that isn't a resource database. I call it a data database (horrible term though that is). Data databases are the normal databases that applications use to store their data. Resource databases store binary code, user interface elements, and other things that make up PalmOS programs (much the same as a Macintosh resource fork). Data databases and resource databases share the same metadata structure, but differ in almost every other respect.

A second complication is PalmOS's category support. Everyone who uses a Pilot is familiar with the category pulldown. Rather than being entirely the application's responsibility, this has operating system support. However, it's only available for data databases, not resource databases, and it's implemented in a rather strange fashion.

Finally, it's important to remember that the data structures are native to the Pilot. As such, they all behave according to the Pilot's rules:

- **Byte** is an unsigned 8-bit byte.
- **Word** is an unsigned 16-bit integer.
- **DWord** is an unsigned 32-bit integer.
- Structure members are aligned on 16-bit, not 32-bit boundaries.
- Everything uses big-endian byte ordering.

The last two, in particular, can give UNIX fits.

## Metadata

The metadata that describes the contents of a data database is divided into three sections: the database information (dbinfo) block, the application information (appinfo) block, and the sort information (sortinfo) block. The appinfo and sortinfo blocks are optional, and are treated as simple bytestrings by the DLP requests that deal with them. Unfortunately, PalmOS has co-opted the appinfo block to hold category names, so that my synchronization application has some more work to do there. I discuss these blocks in later articles.

The dbinfo block is required. It contains information that PalmOS needs to keep track of to deal with the databases. The dbinfo block contains the following information (gleaned from SYSTEM/DLCOMMON.H):

```
Byte name[];
```

Every database has a name. No two databases on the same system should have the same name (although it appears that PalmOS allow two databases on different cards to have the same name). It's a variable-length, zero-terminated (C-style) string.

```
Word dbFlags;
```

This is a bitmask of the following flags:

dlpDBFlagReadOnly	Set if the database is read-only.
dlpDBFlagResDB	Set if the database is a resource database.
dlpDBFlagApp-InfoDirty	Set if the appinfo block has been Modified since the last time the database was backed up.
dlpDBFlagOpen	Set if the database is open (it shouldn't be, by the time the synchronization process is running).
dlpDBFlagBackup	Set if the database should be copied to the desktop if no conduit is defined.

```
DWord type;
```

```
DWord creator;
```

Like the resource/data distinction, PalmOS borrows these two data items from the Macintosh. They identify the application that owns the database. Despite being stored as DWords, they're always presented to users as four ASCII characters. For example, the Memo Pad uses type "DATA" creator "memo". StingerSoft's PilotMoney uses type "MoDT" creator "Mony". It's important to note that these two items are case sensitive, and that creator names in all lowercase are reserved for Palm.

```
Word version;
```

This is an integral version number that applies to the database as a whole. Its primary purpose is to provide applications a way to recognize databases created by older versions of themselves.

```
DWord modNum;
```

This is a count of the number of times that the database has been modified. It's initialized to one when the database is created.

```
DlpDateTimeType crDate;
```

```
DlpDateTimeType modDate;
```

```
DlpDateTimeType backupDate;
```

These are time stamps of when the database was (respectively) first created, last modified, and last backed up by some synchronization process. DlpDateTimeType is a structure:

```
typedef struct DlpDateTimeType {
    Word year;        // year, including century
    Byte month;       // month: 1-12
    Byte day;         // day: 1-31
    Byte hour;        // hour: 0-23
    Byte minute;      // minute: 0-59
    Byte second;      // second: 0-59
    Byte unused;      // unused - set to null!
} DlpDateTimeType;
```

Note that the Pilot has no concept of time zone (unfortunate for those of us who travel fairly often). Also, UNIX stores time stamps as seconds past the epoch; the conversion from the Pilot's format to UNIX's is a bit annoying.

## Data Databases and Records

In addition to the metadata, data databases contain a set of records. Each record contains some application data, which PalmOS treats as a bytestring. Additionally, PalmOS associates a record ID, a category, and some flags with each record. PalmOS also maintains a record list with each database, allowing an application to sort the database relatively easily by shuffling records around in the list.

Internally, PalmOS maintains the following information for each record:

```
DWord recordID;
```

This is a three-byte value (the high-order byte isn't used) that must uniquely identify the record within its database. No record may have an ID of zero.

## Client or Server?

I can't speak for U.S. Robotics, of course, but it's interesting to speculate on why they chose to make the Pilot the HotSync server. Here are three pet theories I've come up with, and my own rebuttals to them:

- Maybe they simply made a design decision, early on, that the Pilot was to be the authoritative repository for all data and the desktop was to be the backup. In that case, why let the desktop do so much data manipulation? It seems like that's exactly the sort of thing they'd want to avoid.
- Perhaps they felt that it was easier to make sure that the server was always present on the Pilot than on the desktop, given that the desktops are running Windows. However, the HotSync Manager itself seems to be always present, and PalmOS isn't exactly designed to make this sort of thing easy. (Amusingly enough, UNIX is designed to make creating persistent daemons easy, unlike the other operating systems involved.)
- Perhaps they simply decided that, since the Pilot is always turned on (and it really is, even though it doesn't seem like it is) and the desktop presumably isn't, the Pilot should be the server. However, the fact that the desktop is persistent in location, and you bring the Pilot to the desktop, suggests that the Pilot should be the client. Also, a lot of desktops (especially in the UNIX world) are left on all the time.

We may never know, and I seriously doubt that this will ever change. But I still have to wonder. If anyone from U.S. Robotics reads this and knows the answer, I'd love to hear it. ✓

Byte attributes;

This is a bitmask of the following flags (from SYSTEM/DATAMGR.H):

dmRecAttrDelete	Set if the record is slated for deletion.
dmRecAttrDirty	Set if the record has changed since it was last backed up.
dmRecAttrBusy	Set if the record is currently in use (it shouldn't be, by the time the synchronization process gets to it).
dmRecAttrSecret	Set if the record is marked "secret," which means that it should be password protected.

Byte category;

This is an integer, ranging from zero to `dmRecNumCategories`, that identifies which category this record is part of.

Word `recSize`;

This is the total size of the record in bytes.

Byte `data[]`;

This is the actual application data.

### *Resource Databases and Resources*

Where data databases contain records, resource databases contain resources. PalmOS maintains a resource list for each resource database, much like it maintains a record list for each data database. However, where data records have a single three-byte ID, resources have a four-character type and an additional numeric ID. (Not surprisingly, this scheme is also borrowed from the Macintosh.)

Internally, PalmOS maintains the following information for each resource:

DWord `type`;

Word `id`;

The type and ID of the resource. Like the database type and creator, the resource type is stored as a DWord, but is always presented to users as four characters (like "code" or "MENU"). Usually, the type and ID are presented together, like "code #0" or "MENU #1000."

Word `resSize`;

This is the total size of the resource, in bytes.

Byte `resData[]`;

This is the actual resource data.

## The HotSync Process

Given all that, we can finally take a quick look at what HotSync looks like at the DLP level. The first thing that happens during a HotSync is a lot of handshaking between the client and the server. Once that's all done, the client starts throwing around a lot of DLP requests that deal with PalmOS databases. Here's the general sequence:

1. The client finds out which memory cards the server knows about, using the `dlpReadStorageInfo` request. It's interesting to note that the current version of the HotSync Manager seems to skip this stuff, presumably because it only supports the current Pilot, which has only one card.
2. For each card, the client finds out which databases are stored on that card, using one or more `dlpReadDBList` requests. Note that this is a per-card operation. Internally, PalmOS maintains separate information for each memory card.
3. If the card contains a database that the client is interested in, it opens the database, using the `dlpOpenDB` request.
4. The client reads data from (or writes data to) the database it just opened, using any of several DLP requests.
5. The client closes the database, using the `dlpCloseDB` request.
6. If there are more databases on this card that the client is interested in, it repeats from step 3.
7. If there are more cards in the system, the client repeats from step 2.

That's all there is to HotSync. The details I gloss over in this quick overview are enough to keep us busy for some time to come.

## Next Issue: Into the Protocol Stack

Next issue, I discuss the protocol stack, and see about writing some code to actually talk to a Pilot. Happy hacking till then. ✓

---

## Disks or No Disks?

PalmOS's database paradigm is a significant departure from the file system paradigm that most other machines use.

Machines with disks tend to keep persistent data in files on disk and treat their RAM as pure scratchpad. This state of affairs came about because, historically speaking, desktop machines haven't done I/O very well. Since it was expensive to pull data off disk into RAM, computing evolved a paradigm of reading data from disk into RAM, modifying them in RAM, then writing them out to disk. Since this read-modify-write cycle necessarily involves I/O (and hence overhead), desktop machines tend to have one format for data on disk, and a separate format for the same data in RAM.

On the Pilot, with no disks, PalmOS applications have to keep their persistent data in RAM. There's no reason for the read-modify-write cycle – where would you read from and write to? – so PalmOS applications just modify their persistent data in place. This also implies that there's only one format for the data.

Additionally, applications need some way of organizing their persistent data. Disk-based systems usually organize data into files, which map well to the read-modify-write paradigm: they open a file, read a whole file into memory (since I/O is expensive, you may as well do a lot of it once, and get it over with), process the data, and write the whole file back out. PalmOS applications don't use the read-modify-write paradigm, so they don't use files. Instead, they use a construct called a PalmOS database that's designed to allow direct access to the data in memory.

It's interesting to note that the read-modify-write file paradigm didn't always exist. Some 35 years ago, the MULTICS operating system was based entirely on the idea that the disk could be mapped directly into memory, giving a disk-based system a set of paradigms that probably wasn't that dissimilar to the PalmOS. However, the hardware of the day wasn't up to it, the simpler file-based idea found its way into CP/M and UNIX, and only recently has the idea of memory-mapped disk found its way back into any modern operating systems. ✓