

UNIX and Pilots Part III: The Minimal HotSync

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

In the last issue of *Handheld Systems*, I focused on the protocol stack and, specifically, the interaction between a Pilot and a desktop system, plus ways to convince a UNIX machine to speak SLP. This time around, I extend that system to perform the most basic synchronization operation possible, the Minimal HotSync. This HotSync operation doesn't actually synchronize any data. Rather, it establishes communications between the Pilot and the desktop, then immediately shuts the link down.

The Minimal HotSync may sound like a very odd thing to implement, but it's actually an extremely useful part of developing any communications system. Successful execution of the Minimal HotSync indicates that the SLP communications framework is functional all the way from the Pilot to the desktop and back again. Until that is determined, debugging and development is pointless, since the framework must be functional before anything else can work.

I implement the Minimal HotSync directly on top of the SLP layer; the PADP and DLP layers are not implemented first. This isn't desirable for a real HotSync, but it is appropriate for testing purposes.

HotSync can be divided into three phases: Establishment, Synchronization, and Termination.

- During Establishment, the Pilot and the desktop go through a handshake sequence to negotiate the particulars of the actual serial connection. The Establishment phase uses the Connection Management Protocol (CMP).
- During Synchronization, the Pilot and the desktop actually exchange data, using the Desktop Link Protocol (DLP). Most of the activity of a real HotSync occurs in this phase.
- Finally, during Termination, the Pilot and the desktop go through another handshake sequence to halt synchronization activity. The Termination phase, like the Synchronization phase, uses DLP.

The Minimal HotSync goes through the complete Establishment phase, skips the Synchronization phase entirely, and goes through the complete Termination phase. To that end, a detailed look at the Establishment and Termination phases is in order.

Establishment

In the initial stage of the Establishment phase, the Pilot spews a stack of about ten Loopback test packets:

```
Pilot: SLP: 3 -> 3 Loopback length 0 XID 0x5E
```

These packets can be ignored since current implementations of HotSync disregard them. After that, the Pilot settles down a bit and sends a CMP Wakeup packet:

```
Pilot: SLP: 3 -> 3 PADP length 14 XID 0xFF
0000 01 C0 00 0A 01 00 01 00 00 00 00 00 E1 00
.....
```

Breaking down this dump in more detail, the following emerges:

```
SLP:          This is a SLP packet.
3 -> 3        It was sent from SLP socket 3 to SLP socket 3,
               which the Pilot manual calls the Desktop Link
               Server socket.
```

```
PADP          The data in this SLP packet is actually a PADP
               packet.
length 14      The PADP packet is 14 bytes long.
XID 0xFF       The transaction ID of this SLP packet is 0xFF.
```

The hex dump actually shows the bytes of the PADP packet. In order to decipher it, first look at the format of a PADP packet. A PADP packet consists of a PADP header followed by some data:

```
[ PADP header ][ Packet body (user data) ]
```

Since PADP packets are encapsulated in SLP packets, and SLP already provides a CRC error check, PADP doesn't provide any error checking.

The PADP header is very simple:

```
typedef struct PadHeaderType {
    Byte type;           // Packet type
    Byte flags;          // Flags
    Word sizeOrOffset;    // Size of data, or fragment offset
} PadHeaderType;
```

The `type` field identifies what kind of PADP packet we're dealing with:

```
#define padData 1 // PADP data packet
#define padAck 2 // PADP ACKnowledgement packet
#define padTickle 4 // PADP Tickle packet
```

`padData` identifies packets carrying ordinary data. `padAck` packets are used to acknowledge receipt of `padData` packets (remember, PADP provides reliability). `padTickle` packets are used for timeout prevention.

The `flags` field and the `sizeOrOffset` field are very important for dealing with PADP's fragmentation feature. I'll discuss these in a later article. For now, simply note that the `flags` field must always be set to `0xC0`, in which case the `sizeOrOffset` field contains the size in bytes of the PADP packet body.

Here are the CMP Wakeup packet contents:

```
01      Type: padData
C0      flags
00 0A   size: the payload is 10 bytes long.
```

That leaves the following for the packet data:

```
01 00 01 00 00 00 00 00 E1 00
```

This must, in turn, be broken down according to the CMP specification. It should come as no surprise that the CMP protocol specifies a header and a body:

```
[ CMP header ][ Packet body (user data) ]
```

Since all the CMP packets have the same header and body structure, PalmOS wraps up the header and body in a single `typedef`:

```
typedef struct CmpBodyType {
    Byte type;           // CMP packet type (header)
    Byte flags;          // flags (body)
    DWord commVersion;   // comm software version (body)
    DWord baudRate;      // serial line speed (body)
} CmpBodyType;
```

`type` defines the type of the packet:

```
#define cmpWakeup 1 // CMP Wakeup packet sent from
                    // server to client
#define cmpInit 2 // CMP Init packet sent from
                  // client to server
#define cmpAbort 3 // CMP Abort packet sent from
                   // client to server
```

The first byte of the CMP packet identifies it as a CMP Wakeup packet. In a Wakeup:

```
flags           Always zero.
commVersion     The server software version.
baudRate        The maximum rate the server supports.
```

The complete breakdown for the CMP Wakeup packet body is:

```
01           type: CMP Wakeup
00           flags: always zero in Wakeup
01 00 00 00  commVersion: we'll call this 1.0.0.0
00 00 E1 00  baudRate: 0x0000E100 == 57600 bps
```

This indicates that the Pilot is announcing its intention to HotSync using version 1.0.0.0 of the HotSync protocol at up to 57600 bits per second.

It's instructive to look back over all the data that actually had to be transmitted for this little bit of information:

```
0000 BE EF ED 03 03 02 00 0E FF AF 01 C0 00 0A 01 00
.....
0010 01 00 00 00 00 00 E1 00 82 D7
.....

BE EF ED          SLP: signature
03               SLP: destination port
03               SLP: source port
02               SLP: next higher protocol
00 0E            SLP: body size
FF               SLP: transaction ID
AF               SLP: header checksum
01              PADP: packet type
C0              PADP: flags
00 0A           PADP: body size
01              CMP: packet type
00              CMP Wakeup: flags
01 00 00 00     CMP Wakeup: protocol version
00 00 E1 00     CMP Wakeup: maximum speed
82 D7           SLP: CRC
```

Note the way the different protocols nest inside each other. This is why it is called a protocol stack.

Back to the Establishment phase. After receiving a valid PADP packet, the desktop must acknowledge it with a `padAck` packet:

```
Desktop: SLP: 3 -> 3 PADP length 14 XID 0xFF
0000 02 C0 00 0A      ....
```

If you break down the PADP packet in the SLP body, you find:

```
02           type: padAck
C0           flags
00 0A        size: the payload is 10 bytes long
```

However, there aren't ten bytes of payload here. At first glance, this seems to be an error. It's all right though because in a `padAck` packet, the PADP `sizeOrOffset` and the SLP transaction ID must be the

same as they are in the packet being acknowledged. A packet with ten bytes of acknowledged payload translates into ten bytes of received payload.

After acknowledging receipt of the CMP Wakeup, send either a CMP Init or a CMP Abort. In an Init:

```
flags           A bitmask identifying changes from the Wakeup.
commVersion     identifies the client software version.
baudRate        specifies the rate the client is going to use.
```

The defined bit values for `flags` are:

```
#define cmpInitFlagChangeBaudRate 0x80
// client changed speed
```

Note that the client may send values of `0x00000000` for `commVersion` and `baudRate` to indicate "whatever the server is using right now." Note also that the initial negotiation in the Establishment phase happens at 9600 bps and, if the client requests a `baudRate` other than 9600 bps, it has to set the `cmpInitFlagChangeBaudRate` bit in `flags`.

The `cmpInit` packet used for the Minimal HotSync looks like this:

```
Desktop: SLP: 3 -> 3 PADP length 14 XID 0xFF
0000 01 C0 00 0A 02 00 00 00 00 00 00 00 00 00
.....
```

```
01              PADP: packet type
C0              PADP: flags
00 0A          PADP: body size
02              CMP: packet type
00              CMP Init: flags
00 00 00 00    CMP Init: client protocol version
00 00 00 00    CMP Init: client desired speed
```

To simplify matters, follow the server's recommendations.

The client is allowed to refuse to speak to the server by sending a CMP Abort packet. In an Abort:

```
flags           specifies the reason for the abort.
commVersion     is always zero.
baudRate        is always zero.
```

Currently, a mismatch between `commVersions` is the only supported reason for an Abort:

```
#define cmpAbortFlagVersionError 0x80
```

After the Pilot receives the CMP Init (or CMP Abort), it has to acknowledge receipt with a `padAck` packet. Once the desktop receives the `padAck` packet, the Establishment phase is over.

Synchronization

The Synchronization phase, which will be covered in detail in a later article, consists of a series of DLP requests and responses. The Minimal HotSync skips it entirely.

Termination

The Termination phase consists of a single DLP request, `dlpEndOfSync`, sent from the client to the server.

DLP packets ride inside PADP packets, just like CMP packets, and have a header and a body, like CMP packets. However, DLP is a bit more complex than CMP. First, DLP requests have a slightly different header format than DLP responses. Second, the body of a DLP packet is broken down into zero or more arguments:

[DLP header][DLP arg 1][...]

Here's what the DLP request and response headers look like:

```
typedef struct DlpReqHeaderType {
    Byte id;        // request function ID
    Byte argc;      // count of args that follow this header
} DlpReqHeaderType;

typedef struct DlpRespHeaderType {
    Byte id;        // response function ID
    Byte argc;      // # of arguments that follow this header
    Word errorCode; // error code
} DlpRespHeaderType;
```

The high-order (0x80) bit of the `id` differentiates between a request packet (clear) or a response packet (set). The low-order seven bits identify the particular DLP transaction involved. Additionally, the `id` is the way to distinguish between DLP and CMP. DLP request ids start with 0x10, where CMP uses 0x01, 0x02, and 0x03.

`argc` specifies how many arguments the request or response carries with it. For a response, `errorCode` carries out-of-band status information. 0 is defined to mean no error; non-zero values all indicate different error codes. Error codes less than 128 are reserved by Palm.

Arguments consist of an argument header followed by data. There are two different types of arguments, small and big (really, I'm not making this up). Hence, there are two different types of argument headers. Each header gives an ID and a size for the argument.

In the PalmOS headers, the argument headers and argument data are wrapped up in a fairly ugly set of unions, because of the way all the packing and alignment issues fall out. In this case, a couple of simpler structures that illustrate how the serial stream should be interpreted are defined.

```
typedef struct {
    Byte id;        // Argument ID
    Byte size;      // Argument data size
} DlpSmallArgHeader;

typedef struct {
    Byte id;        // Argument ID
    Byte unused;    // Always zero
    Word size;      // Argument data size
} DlpBigArgHeader;
```

Small arguments can be 0 - 255 bytes long, while big arguments can go up to 64 KB. The high bit (0x80) of the argument `id` indicates the difference; it's clear for small arguments and set for big arguments.

The `dlpEndOfSync` packet appears as follows:

```
Desktop: SLP: 3 -> 3 PADP length 4 XID 0xFF
0000 01 C0 00 06 2F 01 20 02 00 00      ..../. ...
```

```
01          PADP: packet type
C0          PADP: flags
00 06      PADP: body size
2F         DLP: request 0x2F
01         DLP: argc
20         DLP: small arg #1 ID 0x20
02         DLP: small arg 0x20 size
00 00      DLP: small arg 0x20 data
```

Request ID 0x2F is `dlpEndOfSync`. For `dlpEndOfSync`, argument ID 0x20 is called `termCode` and identifies the reason for ending the synchronization. `termCode` 0 is `dlpTermCodeNormal`, meaning all's well.

After receiving the `dlpEndOfSync` request, the Pilot must acknowledge receipt with a `PadAck` packet. After acknowledging, it's must also send a `dlpEndOfSync` response packet:

```
Pilot: SLP: 3 -> 3 PADP length 8 XID 0xFF
0000 01 C0 00 04 AF 00 00 00      .......
```

```
01          PADP: packet type
C0          PADP: flags
00 04      PADP: body size
AF         DLP: response to 0x2F
00         DLP: argc
00 00      DLP: errorCode
```

This is a response to `id` 0x2F, `dlpEndOfSync`, carrying no arguments, reporting a status code of 0, all's well. This is precisely the outcome desired. Once observed, send a `PadAck`, and you're finished.

Putting it all together, here's the sequence of packets that's needed:

```
Pilot:
SLP: 3 -> 3 Loopback length 0 XID 0x5B
```

(The above might appear many times or none at all.)

```
Pilot (CMP Wakeup):
SLP: 3 -> 3 PADP length 14 XID 0xFF
0000 01 C0 00 0A 01 00 01 00 00 00 00 00 E1 00 xx xx
.....
```

```
Desktop (ACK):
SLP: 3 -> 3 PADP length 14 XID 0xFF
0000 02 C0 00 0A
....
```

```
Desktop (CMP Init):
SLP: 3 -> 3 PADP length 14 XID 0x01
0000 01 C0 00 0A 02 00 00 00 00 00 00 00 00 00
.....
```

```
Pilot (ACK):
SLP: 3 -> 3 PADP length 4 XID 0x01
0000 02 C0 00 0A
....
```

```
Desktop (dlpEndOfSync request):
SLP: 3 -> 3 PADP length 4 XID 0x02
0000 01 C0 00 06 2F 01 20 02 00 00
..../. ...
```

```
Pilot (ACK):
SLP: 3 -> 3 PADP length 4 XID 0x02
0000 02 C0 00 06
....
```

```
Pilot (dlpEndOfSync response):
SLP: 3 -> 3 PADP length 8 XID 0x03
0000 01 C0 00 04 AF 00 00 00
.....
```

```
Desktop (ACK):
SLP: 3 -> 3 PADP length 8 XID 0x03
0000 02 C0 00 04
....
```

The Minimal HotSync implementation

Given all that, take a look at the implementation for the Minimal HotSync program; a simple state machine. The transitions look like this:

Start: WAITING_FOR_WAKEUP

```
WAITING_FOR_WAKEUP:
    receive a packet
    CMP Wakeup -> ACK_WAKEUP
    anything else -> WAITING_FOR_WAKEUP
```

```
ACK_WAKEUP:
    send padAck
    -> SEND_CMP_INIT
```

```
SEND_CMP_INIT:
    send CMP Init
    -> WAITING_FOR_INIT_ACK
```

```
WAITING_FOR_INIT_ACK:
    receive a packet
    PadAck -> SEND_END_OF_SYNC
    anything else -> WAITING_FOR_INIT_ACK
```

```
SEND_END_OF_SYNC:
    send dlpEndOfSync request
    -> WAITING_FOR_END_OF_SYNC_ACK
```

```
WAITING_FOR_END_OF_SYNC_ACK:
    receive a packet
    PadAck -> WAITING_FOR_END_OF_SYNC_RESPONSE
    anything else -> WAITING_FOR_END_OF_SYNC_ACK
```

```
WAITING_FOR_END_OF_SYNC_RESPONSE:
    receive a packet
    dlpEndOfSync response
    -> ACK_END_OF_SYNC_RESPONSE
    anything else -> WAITING_FOR_END_OF_SYNC_RESPONSE
```

```
ACK_END_OF_SYNC_RESPONSE:
    send PadAck
    done!
```

In the actual Minimal HotSync implementation (called MinSync), the necessary packets are statically defined to send and receive an array of Bytes. A send operation in the state machine is a single function call, since the bytes to send are precomputed. A receive operation is a function call to receive a SLP packet, then a byte-compare to see if it's the right one. Obviously, this is a tremendously inflexible approach, but it is simple, which makes it ideal for MinSync.

The state machine is primarily implemented with a state variable and a big switch statement. I collapsed the WAITING_FOR_..._ACK states into a single logic block that actually runs before the switch statement simply to reduce the number of lines of code. This complicates the state machine somewhat, but it's still fairly straightforward. MinSync also needs to allocate a UDP socket and send a Register packet to the SLPd before it can do any SLP at all. This is only a few lines of code and it is very straightforward.

If you run MinSync and press the HotSync button on your Pilot, you should see and hear the Pilot go through a very brief, but correct, HotSync. A modest beginning, but a very important first step.

The code for MinSync is on the Web at <http://www.kodachi.com>. It should also be on the source disk for this issue of *Handheld Systems*. Next time, I focus on a real PADP layer. See you then. ✓