

## UNIX and Pilots: The Protocol Stack, Part 1

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

Copyright © 1997 by Kevin L. Flynn. All rights reserved.

Last issue, I focused on HotSync and Pilot background issues. This issue, it's time to go diving into the protocol stack and take an initial look at the UNIX-side implementation.

### The Protocol Stack

For two computers to communicate, they must agree on the precise meaning of every byte they send to each other. In practice, this is accomplished by specifying a set of rules that each computer obeys. The set of rules is known as a computer communications protocol, or simply a protocol.

Creating accurate protocol specifications is difficult. On one hand, we need to support extremely complex, high-level operations. On the other hand, the computer is so stupid that we have to specify each little detail of every step to be carried out. There's a significant conflict here.

Fortunately, the problem is not insurmountable. It's possible to implement a simple protocol, then use it as a building block for a more complex protocol and, in turn, use that protocol to generate still more complex protocols. This process is called layering, or stacking, in protocol design, referring to the way different protocols ride atop one another. A complete set is called a protocol stack.

Although this article isn't a course on protocol design, it's worthwhile to look briefly at two major classifications among computer communications protocols. First, protocols fall into two general categories: datagram and stream. In a datagram protocol, the data is divided into a number of discrete chunks, called datagrams, presumably after telegrams. In contrast, a stream protocol does not include discrete chunks; the data just keeps coming with no particular boundaries. The choice between a stream protocol and a datagram should be based on which one fits your application's use better.

Second, protocols are generally reliable or unreliable. A reliable protocol ensures that data gets to its destination, or informs you that it did not. An unreliable protocol guarantees only that it will try to get your data to its destination, but anything can happen. Though unreliable protocols may appear useless, in fact they are quite handy. They can offer significant performance advantages over reliable protocols in some applications. Moreover, it's always possible to build a reliable protocol over an unreliable protocol if necessary.

Given that background, let's look at HotSync's protocol stack. As we do, bear in mind that this certainly isn't the only way U.S. Robotics could have done it, nor is it necessarily the best (a lot of us wonder why they didn't just use TCP/IP). It works, though.

HotSync's stack consists of four separate datagram protocols in three layers:

Layer 3	DLP	CMP
Layer 2	PADP	
Layer 1	SLP	

(Layer numbers increase with the progression from simpler lower-level protocols to more complex higher-level protocols.)

### DLP

One of the two protocols in the top layer is DLP, the Desktop Link Protocol. In the last issue of *Handheld Systems*, I included a brief glimpse at DLP, which defines a set of requests that the HotSync client can make to the server and a set of corresponding responses that the server can hand back to the client. All the overt actions of HotSync happen as DLP requests to transfer data back and forth.

Remember that DLP is a datagram protocol – each DLP request and response rides in its own datagram. In turn, these datagrams are handed down to layer two (PADP) as input.

### CMP

Occupying the other slot in the top layer is CMP, the Connection Management Protocol. CMP packets are used at the start of the HotSync process to negotiate speed and protocol version. CMP is also the location of the single most annoying aspect of the whole system – the connection negotiation at the start of a HotSync session is initiated by the Pilot rather than the desktop, completely reversing the Pilot-as-server concept for one short exchange of packets.

### PADP

Layer two holds PADP, the Packet Assembly-Disassembly Protocol. PADP provides two crucial features: fragmentation and retransmission.

Fragmentation involves taking a large datagram and splitting it into chunks. Each chunk is handed to a lower-level protocol for transmission. On the other end, all the chunks are reassembled back into the complete large datagram. This is an important function because lower-level protocols can often be simplified by imposing a fairly low limit on the size of a datagram. (Also, recall that we're working with a serial line here, maybe a modem – it's slow. Keeping the low-level datagrams small gives the system a better chance of responding quickly to user requests or odd events.)

Retransmission addresses the need for some reliability in a protocol stack, but modems and serial lines are not guaranteed to be perfect.

They may damage datagrams or drop them entirely. PADP defines a mechanism, called positive acknowledgment, to acknowledge receipt of data and to retransmit datagrams that aren't acknowledged within a certain amount of time. Positive acknowledgment is a common technique for building a reliable protocol on top of an unreliable protocol.

### SLP

Finally, at the bottom of the stack is SLP, the Serial Link Protocol. SLP is an unreliable datagram protocol that provides four features: framing, integrity checks, packet ordering, and a port space.

#### Framing

Datagram protocols, by definition, operate on bounded chunks of data. However, SLP runs over RS232, which isn't datagram-oriented. This means that SLP must provide framing, a mechanism to detect datagram boundaries.

#### Integrity checks

In the world of computer communications, imperfections manifest as data changing as it is sent from one system to another. This might be as subtle as a single inverted bit or as extreme as a transmission shutdown. In any case, it's absolutely crucial that some mechanism detect and correct errors in transmission.

The most common error-checking mechanism is the checksum, which is literally the sum of the different bytes of data. We can use this idea as follows:

- 1) Sender gets a datagram to send.
- 2) Sender calculates the checksum.
- 3) Sender sends the datagram and the checksum to the recipient.
- 4) Recipient calculates the checksum.
- 5) Recipient compares its checksum to the one sent by the sender.
- 6) If they differ, something went wrong.

The problem is that checksums are not adept at detecting errors. As an example, suppose that the three bytes 0x44 0x55 0x66 (checksum 0xFF) are sent, but the network mangles them to 0x45 0x54 0x66. This isn't what we sent, but 0x45 + 0x54 + 0x66 is still 0xFF, so the checksums match even though the packet is corrupted.

To deal with this problem, the family of Cyclic Redundancy Check (CRC) codes was invented. Basically, CRC codes cycle through each bit of the input data in such a way that the CRC value changes even for subtle differences like transposed bits. The trade-off, of course, is that it's more computationally intensive to calculate a CRC than to calculate a checksum. However, in most situations it's worth it.

### Packet ordering

A common failure is that in many networks entire packets are transposed or duplicated. RS232 serial links don't typically have this problem. However, bugs in the software implementing higher-layer protocols can cause these problems. As a result, SLP must provide a mechanism to determine when packets are received out of order or more than once.

### Port space

Most networks have to contend with the fact that there is only one network interface (the serial cable, in this case), but there are several processes that want to use the interface. While it's no big trick to allow multiple processes to stuff bytes onto the serial cable, it's a totally different matter to figure out which process should get the bytes that the operating system just read off of the cable.

A common solution is to associate a small number, usually called a port number or socket number, with each packet. Processes tell the operating system which port numbers to use, and the operating system then uses the port number to demultiplex packets as they come in.

## SLP's Mechanisms

SLP provides all these things in relatively simple ways. An SLP datagram looks like this:

```
[ SLP preamble ][ SLP header ]
[ Packet body (user data) ][ SLP CRC ]
```

The preamble is SLP's framing mechanism; every packet transmission begins with the three-byte preamble 0xBE 0xEF 0xED, enabling receivers to easily locate the start of the packet. The header, which is always seven bytes long, follows the preamble:

```
Byte dest      // Destination socket number
Byte src       // Source socket number
Byte type      // Type of packet
Word bodySize  // Size of body
Byte transId   // Transaction ID
Byte checksum  // Checksum of header
```

The `dest` and `src` fields identify the socket the packet was sent from and the socket it's being sent to, respectively. Both are present because you need to know the source socket to be able to reply to a packet. These fields are eight bits wide, implying that SLP supports 256 sockets.

The `type` field identifies the higher-level protocol where the packet originated. It can have one of three values:

```
#define slkPktTypeSystem 0    // System packets
#define slkPktTypePAD    2    // PAD Protocol packets
#define slkPktTypeLoopBackTest 3 // Loop-back test packets
```

(Packet type 1 was previously used for CMP, but is not any longer.)

Practically everything I discuss uses PADP packets and type `slkPktTypePAD`. Loopback packets appear early in the HotSync process, but can be ignored. System packets, however, are used exclusively for the remote debugger, which isn't the focus of this series (although it may come up later).

The `bodySize` field contains the size of the packet body, in bytes. SLP limits packets to 1024 bytes of packet body.

The `transId` field is SLP's answer to the packet ordering problem. SLP, however, doesn't actually interpret it, but just passes it on. The idea is that higher-level protocols can (and should) provide each packet with a unique transaction ID for ordering packets and detecting duplicates. In effect, SLP gives higher-level protocols a place to put the transaction ID.

Finally, the `checksum` field is an eight-bit simple checksum of the entire preamble and header, not including the checksum byte itself.

After the header comes `bodySize` bytes of packet body, typically data from higher-level protocols, then a sixteen-bit CRC of the preamble, header (including the checksum byte this time), and packet body. (See "Error Detection and Recovery" on page 32 for more on how this is used.)

## SLP on UNIX

Since SLP is at the bottom of the protocol stack, it's a natural place to start adapting HotSync to a UNIX machine. However, since SLP isn't the only thing in the protocol stack, it's worth taking a moment to think about how to make it easier to fit the rest of the stack together.

The task involves thinking of SLP as a black box, and looking at the things at its edges. A connection to the Pilot should be on one side of the SLP box; on the other should be a connection to higher-level protocols. The connection to the Pilot is a serial port over which we're reading and writing properly-formatted SLP packets.

The connection to the higher-level protocols is more problematic. An obvious idea is to make that connection a group of C functions, then construct higher-level protocols in the form of a set of calls to those functions. However, it's remarkably difficult to reconcile this approach with UNIX's I/O model.

The problem is that multiple independent processes need simultaneous access to SLP, without any particular synchronization with each other. In this situation, the UNIX I/O model specifies individual file descriptors for each process on some SLP device, file, or resource. However, that model breaks down using libraries because file descriptors in UNIX must refer to UNIX devices, and making user-level code look like a UNIX device is impossible. Though it is possible to code around the problem, it gets fairly ugly (it's easier if your UNIX is multithreading, but not all of them are).

Given that we can't make user-level code look like a UNIX device, the options are to write a UNIX device driver or to figure out a way to make the UNIX kernel handle the necessary multiplexing and demultiplexing. Device drivers aren't very portable, so it makes more sense to get the kernel to do the right thing at the user level.

Using UNIX's networking support, I can write a user-level daemon that translates between SLP packets on one side and UDP/IP packets on the other. UDP/IP is an unreliable datagram protocol from the Internet protocol suite. Creating a simple mapping between SLP and UDP is easy, and solves the multiple-access problem nicely, since any number of peers can send and receive data from a UDP port at any given moment. The peers can be UNIX processes, threads within UNIX processes on systems that support threads, or whatever. They don't even have to reside on the same machine as the server, although the usefulness of that in this case is far from obvious.

### SLPD

My first program is called SLPD, the SLP daemon. Conceptually, it's relatively simple; it copies SLP packets between a serial port and a UDP socket. However, even this relatively simple program is rather complex in the UNIX environment.

The first complication is once again due to the UNIX I/O model, which is fully blocking to better support multiple users. I need to watch two different file descriptors, one for the serial device and one for the UDP socket. That always complicates matters in single-threaded UNIX. I address this issue by building SLPD on a fairly general I/O library that I wrote some time ago.

The second complication is keeping track of which peer is interested in which SLP socket. Four separate types of UDP packets come into play:

- 1) A peer sends a REGISTER packet to let SLPD know that the peer is interested in a particular SLP socket.
- 2) A peer sends an UNREGISTER packet to reverse a REGISTER packet.
- 3) SLPD sends a STATUS packet back to a peer to tell it whether a REGISTER or UNREGISTER succeeded. This is necessary because we only want one peer to be able to register for each SLP socket, and unregistering for a socket that the peer doesn't register is a little silly.
- 4) Finally, peers and SLPD both send DATA packets containing SLP packets.

Implementing the packets is exceedingly simple: the first byte is either "R", "U", "S", or "D" to indicate REGISTER, UNREGISTER, STATUS, or DATA, respectively. The rest of the packet is dependent on which of the above is selected. REGISTER and UNREGISTER are followed by a single byte containing an SLP socket number. STATUS is followed by a word (two bytes) of status information. DATA is followed by an SLP header and body, including the checksum and CRC.

SLPD drops packets that don't have good checksums or CRCs; there's no point in forwarding bad packets. Additionally, STATUS is only used for the REGISTER and UNREGISTER packets, since those must be reliable. Since SLP is itself unreliable, never use STATUS for DATA packets.

That deals with the UDP side of SLPD. What about the SLP side (which is the point of the whole program, after all)? Here, a few major issues emerge: framing, data formats, and the actual serial line.

Output framing is trivial, requiring only an assurance that the SLP preamble is prepended and a proper CRC appended to each output packet. Input framing, however, is considerably more obnoxious.

The difficulty is that the serial line is not datagram oriented. As such, you can't just instruct UNIX to "go read a SLP packet." Instead, you must read as much as possible into a buffer, then go hunting for a SLP preamble. You can throw away anything before the first preamble in the buffer, since the preamble marks the first useful data.

Given a preamble, you must make sure that the buffer includes an entire SLP header which may require reading additional data. Given a header, it's possible to verify its checksum and look at its `bodySize` field to discover the amount of data to expect. The `bodySize` allows you to read the body data, if it's not already present in the buffer (which is possible, if the packet is small). After that, you can read and verify the CRC, and the entire packet is complete.

Data format issues get wrapped up in all this: everything going over the wire to or from the Pilot must adhere to the Pilot's rules for data formats. Briefly restated, that is eight-bit Bytes, 16-bit big-endian Words, and 32-bit big-endian DWords, with Words and DWords aligned on 16-bit boundaries. (If that doesn't make any sense, read "Data Formats" on page 32.)

Most UNIXes these days have an eight-bit unsigned `char`, a 16-bit unsigned `short`, and a 32-bit unsigned `int`. These map well to the Pilot data types, so it isn't too difficult to operate on Pilot data elements in native UNIX code. That's the good news. The bad news is that the endianness and alignment issues get kind of thorny.

## Endianness

Endianness is especially tricky since UNIX runs on both big and little-endian machines. Byte-swapping is clearly necessary on little-endian machines and unnecessary on big-endian machines. There are two ways to manage this.

The first approach takes advantage of the fact that the Internet protocol designers encountered the same problem quite some time ago. They addressed it by defining network byte order as big-endian and requiring developers to use network byte order in places where it matters. As such, the Berkeley sockets APIs provide functions to translate between host byte order and network byte order. Since every modern UNIX provides the sockets API, these functions are available for use.

The second approach capitalizes on the fact that UNIX I/O always involves streams of bytes. As such, you can always manipulate the bytes directly by simply pulling out one byte at a time and shifting it into

native Words or DWords. For example, a big-endian DWord read from a file descriptor into a Byte array works regardless of the endianness of the processor it's running on:

```
DWord grab_a_dword( int fd )
{
    Byte buf[4];
    DWord dw;
    read( fd, buf, 4 ); /* Ignore error checking for now */
    dw = ( buf[0] << 24 ) | (buf[1] << 16) |
        ( buf[2] << 8 ) |  buf[3];
    return( dw );
}
```

This always works because the left-shift operator is defined to do the correct thing with either endianness, and the order of the bytes in `buf[ ]` isn't affected by endianness – the bytes come over the wire in big-endian order.

## Alignment

UNIXes tend to be more restrictive about data alignment than the Pilot. The Pilot is concerned with code density and power consumption, while the UNIX machine is really only concerned with speed. As such, most UNIXes want data types aligned according to their size. Although this causes no problem for the 16-bit `short` aligned on a 16-bit boundary, the 32-bit `int` aligned on a 32-bit boundary confuses things. Further, you can't use a `realign` function call, since sliding things up in memory to properly align them quickly becomes very ugly.

Once again, there are two ways to deal with this problem. First, most UNIX compilers can generate code that assumes only 16-bit alignment, instead of assuming that 32-bit data is aligned on 32-bit boundaries. However, this proves a maintenance and portability nightmare (especially since there is no real standard way of telling the compiler to do this). Furthermore, the code is slower than necessary.

The other way is to manipulate the bytes directly. Although this is slightly more annoying to write, it's much less fraught with peril than the other option.

## Serial line issues

One more detail here is that UNIX was designed with the underlying assumption that there would be Teletype Model 33s attached to its serial ports, which humans would use to log in. As such, UNIX traditionally sets the serial ports to 300 bps, 7 data bits, 1 stop bit, and even parity. Then UNIX runs some code called a line discipline to actually read and write characters to the serial port. The line discipline performs functions such as mapping carriage return to newline, handling the backspace and delete keys, ensuring that control-C interrupts the running process, and so on.

While all this is handy for humans on Teletypes, it confuses the Pilot tremendously. Consequently, care must be taken to instruct UNIX to set the serial port to 19,200 bits per second, eight data bits, one stop bit, parity and line discipline disabled.

Unfortunately, serial port handling is one of the least portable areas of UNIX and, as such, the single biggest problem area for porting SLPD. I use POSIX `termios` to handle the serial line. If your UNIX doesn't support `termios`, you'll have to do a little work to get my code running.

## The Code

That's enough information to assemble the SLPD. I present pseudocode for the daemon on pages 33-34. The full C source code can be found on the source code disk for this issue of *Handheld Systems*, or on the Web at <http://www.kodachi.com/>. Note that the code has only been tested on Sun Solaris 2.5, though it has previously been tested on both SPARC and x86 hardware. If you have problems with other platforms, please let me know by sending e-mail to [flynn@kodachi.com](mailto:flynn@kodachi.com).

Next time: The Minimal HotSync. Get a head start if you like – the source code is already on the Web. See you then. ✓



## Error Detection and Recovery

In a perfect world, every SLP packet comes through clean and all the checksums match. But what happens in our imperfect world? What types of failures occur?

An obvious failure is if the checksum and CRC do not match. A more subtle failure might be preamble corruption. In that case, the receiver would never notice the packet and it would vanish down the wire. An even more subtle failure is a corrupted preamble on a packet in which the body contains the three bytes `0xBE 0xEF 0xED`. The receiver would treat the preamble sequence in the body as a real preamble and try to interpret the packet it thought it saw. Of course, it is highly unlikely that the checksum and CRC of the false packet will match, but it's a nonzero chance, in which case a completely spurious packet would be sent to some unsuspecting higher-level protocol.

SLP deals with all of these failures (and every other kind of failure) in exactly the same manner. Any time it detects anything unusual about the current packet, it simply stops processing that packet and looks for a new one. No higher-level protocol is alerted or correction attempted; the erroneous packet is simply dropped on the floor. Remember that SLP is an unreliable protocol. Since the higher-level protocols have to be prepared to deal with packets getting dropped anyway, why should SLP try to do anything other than throw away packets it can tell are bad?

Of course, the spurious-packet scenario presented a moment ago still looms. The higher-level protocols must figure out a way to cope with that. The operating system can help by discarding packets addressed to a socket that no application is interested in. Other than that, it's up to the higher-level protocol.

### Data Formats

Every computer has to make several design trade-offs in regard to data storage and access. Two fundamental trade-offs involve endianness and alignment.

Endianness refers to the order of the bytes of a multiple-byte value. (The word comes from *Gulliver's Travels*, by Jonathan Swift, in which a war breaks out over which end of an egg should be broken.) For example, suppose we want to store the 32-bit value `0x12345678` at address `0x1000`. There are two ways to do this. The first way is:

```
0x1000 12 34 56 78
```

This is called big-endian byte ordering, because the `12` (meaning `0x12000000`, a big value) comes first in memory. While it can make the hardware of the processor a little bit more annoying to implement, it's easier to read multiple-byte values in hex dumps.

The other obvious way looks like this:

```
0x1000 78 56 34 12
```

This is called little-endian, because it puts the `78` (meaning `0x78`, a little value) first in memory. Although it is tough to read hex dumps, it is a bit easier to build the hardware.

At this point, the choice of which is better is basically a religious issue – it doesn't really matter, except that you have to know the endianness of data you read. If you get it wrong, `0x1` turns into `0x01000000` – a factor of over 16 million, clearly a problem.

Alignment refers, loosely speaking, to how closely you can pack multiple-byte values in memory. On the Pilot, everything is aligned on 16-bit boundaries, meaning that no multiple-byte value can start on an address that isn't an even multiple of two (16 bits == two bytes). For example, take this structure:

```
struct {
    Byte foo;
    DWord bar;
} foo_struct;
```

Supposing that this structure starts at location `0x1000`, one might

expect it to be laid out as follows:

```
0x1000      foo
0x1001-0x1004 bar
```

However, `0x1001` isn't a multiple of two, so in fact the Pilot lays this structure out like this:

```
0x1000      foo
0x1001      unused
0x1002-0x1005 bar
```

Now `bar` is at `0x1002`, which is a multiple of two, so all is well by the alignment rules. However, we're wasting a byte of memory. Why?

The trade-off here is based on the Pilot's processor hardware, the Motorola 68328 Dragonball. Dragonball hardware cannot access memory on addresses that aren't aligned on a 16-bit boundary, and it prefers to do 16-bit accesses.

If the Dragonball tries to read `bar` from the structure, using the second layout, all is well. It can do a 16-bit read from address `0x1002`, grabbing `0x1002-0x1003` in a single bus cycle, and then follow with a 16-bit read from address `0x1004`, grabbing `0x1004-0x1005`. By contrast, if it uses the first layout, it has to grab `0x1000-0x1001` in one cycle, then `0x1002-0x1003` in a second, then `0x1004-0x1005` in a third, then mix and match bits from all three bus cycles.

The situation actually worsens in the UNIX world. Where the Dragonball is built around a 16-bit external data bus, UNIX boxes tend to use processors built around 32-bit (or even 64-bit) external data buses. As such, where the Dragonball likes addresses aligned on 16-bit boundaries, the processors in UNIX machines tend to like addresses aligned on 32-bit boundaries.

This means that, on most UNIX machines, the structure looks like this:

```
0x1000      foo
0x1001-0x1003 unused
0x1004-0x1007 bar
```

This way, the 32-bit value is properly aligned to be read or written in a single bus cycle. Of course, three bytes are wasted, but UNIX machines tend to have a lot more memory than the Pilot and care substantially more about speed, so it's worth it.

One final note: I've been saying that the hardware cannot operate on addresses that aren't properly aligned. What about Bytes, though (or Words on 32-bit machines)? For instance, take this structure:

```
struct {
    Byte foo;
    Byte bar;
    Word baz;
} foo_struct;
```

If `foo_struct` starts at address `0x1000`, and a 32-bit UNIX machine needs to fetch `bar` and `baz`, what happens?

The answer is that it's not a problem, since eight-bit Bytes and 16-bit Words are smaller than the 32-bit data bus. To fetch `bar` (at `0x1001`), the hardware initiates a bus cycle with address `0x1000` and simply informs the memory hardware that only the second eight bits matter.

Likewise, to fetch `baz` (at `0x1002`), the hardware initiates a bus cycle with address `0x1000` and tells the memory hardware that only the second 16 bits matter. However, `baz` still has to be aligned on a 16-bit boundary. The processor hardware cannot tell the memory hardware to skip eight bits, then hand over the next 16. When dealing with 16 bits, everything must be a multiple of 16. ✓

## SLPD Pseudocode

SLPD is built using a package called FTools (also available at <http://www.kodachi.com/>) which provides several programming tools. In particular, SLPD uses the FTools callback-based I/O library. This library lets a programmer associate read and write routines with UNIX file descriptors. Whenever it's possible to read data from the file descriptor, the read routine is called. If a read routine wants to dump data to one of the other file descriptors, it calls the output descriptor's write routine.

This pseudocode only presents the read and write routines. The FTools code and the code to do REGISTER and UNREGISTER is omitted, as is most error checking and real-world concerns like memory manipulation. However, none of the logic is deliberately wrong. ✓

/\*\*\*\*\*\*Read routine for serial port\*\*\*\*\*

```
static char buffer[8192]
static char *next = buffer;
static int bytes_left = sizeof( buffer );

/*We must read data, even if some left in buffer. */

read up to bytes_left bytes
store them in memory at next
adjust bytes_left to reflect how many bytes are
  unused in the buffer

/* Loop - we might've read multiple packets */

while ( 1 ) {
  ptr=point to 1st occurrence 0xBE 0xEF 0xED in buffer

  if ptr is NULL
    /* No preamble - throw this data away. */

    next = buffer
    bytes_left = sizeof( buffer )

    break

  /* If we drop this packet, don't get confused by
   * this signature again */

  three bytes at ptr = 0

  /* The full header is 10 bytes -
   * do we have all of it in memory? */

  do we have 10 bytes in the buffer _after_ ptr?

  no:
    do we have ROOM for 10 bytes in buffer after ptr?

  no:
    /* Don't have room to read the whole packet.
     * Make room by sliding the good data down to
     * the start of the buffer. */

    slide data from ptr to the end of the buffer
    down to the start of the buffer

    adjust bytes_left

    /* Whether or not we had to do surgery, leave: we
     * have to wait for next callback to read more */

    break
```

/\* Given the header, unpack it. \*/

```
dest =      byte at ptr + 3
src  =      byte at ptr + 4
type  =      byte at ptr + 5
bodySize = word at ptr + 6
transID = byte at ptr + 8
checksum = byte at ptr + 9
```

```
if checksum is bad
  continue
```

/\* The header is ten bytes, then there's bodySize  
 \* bytes of data, then two bytes of CRC.  
 \* Do we have all that in memory? \*/

```
do we have 10 + bodySize + 2 bytes in the buffer
  _after_ ptr?
```

```
no:
  do we have ROOM for bodySize + 12 bytes in the
    buffer after ptr?
```

```
no:
  slide data from ptr to the end of the buffer
  down to the start of the buffer
```

```
adjust bytes_left
```

```
break
```

```
CRC = word at ptr + bodySize + 10
```

```
if CRC is good
  send DATA packet (
    dest, src, type, bodySize, transID, checksum,
    bodySize bytes starting at ptr, CRC
  ) to UDP side
```

```
slide data from ptr + bodySize + 12 to the end of
  the buffer down to the start of the buffer
```

```
adjust bytes_left
```

```
}
```

/\*\*\*\*\*\*Write routine for serial port\*\*\*\*\*

```
char header[10]
```

```
three bytes at header = 0xBE 0xEF 0xED
byte at header + 3 = dest
byte at header + 4 = src
byte at header + 5 = type
word at header + 6 = bodySize
byte at header + 8 = transID
```

```
calculate checksum over first nine bytes of header
```

```
byte at header + 9 = checksum
```

```
calculate CRC over header + body
```

```
write header + body + CRC
```

Read routine for UDP port

```
read packet into buffer

when buffer[0] is R
  do a REGISTER

when buffer[0] is U
  do an UNREGISTER

when buffer[0] = S
  note that we shouldn't have seen it

when buffer[0] = D
  dest =      byte at buffer + 1
  src =      byte at buffer + 2
  type =      byte at buffer + 3
  bodySize = word at buffer + 4
  transID =   byte at buffer + 6
  checksum =  byte at buffer + 7

  if checksum is good
    CRC = word at buffer + 1 + bodySize

    if CRC is good
      send (
        dest, src, type, bodySize, transID, checksum,
        body, CRC
      ) to serial side
```

Write routine for UDP port

```
check the registry for the destination socket

if someone is registered
  char header[bodySize + 13]

  bytes at header = 'D'
  byte at header + 1 = dest
  byte at header + 2 = src
  byte at header + 3 = type
  word at header + 4 = bodySize
  byte at header + 6 = transID
  byte at header + 7 = checksum

  bodySize bytes at header + 8 = body data

  word at header + 8 + bodySize = CRC

  send header to whoever's registered
```