32.4.2 Unpacking a Stack Frame

Requesting a particular byte ordering may be necessary when you work with binary data coming from some specific architecture whereas your program could run on a totally different system. As an example, assume you have 24 bytes containing a stack frame as it happens on an Intel 8086:

```
      +---------+        +----+----+              +---------+
TOS:  |   IP    |  TOS+4:| FL | FH | FLAGS  TOS+14:|   SI    |
      +---------+        +----+----+              +---------+
      |   CS    |        | AL | AH | AX            |   DI    |
      +---------+        +----+----+              +---------+
                         | BL | BH | BX            |   BP    |
                         +----+----+              +---------+
                         | CL | CH | CX            |   DS    |
                         +----+----+              +---------+
                         | DL | DH | DX            |   ES    |
                         +----+----+              +---------+
```

First, we note that this time-honored 16-bit CPU uses little-endian order, and that's why the low order byte is stored at the lower address. To unpack such a (signed) short we'll have to use code `v`. A repeat count unpacks all 12 shorts:

```
my( $ip, $cs, $flags, $ax, $bx, $cd, $dx, $si, $di, $bp, $ds, $es ) =
  unpack( 'v12', $frame );
```

Alternatively, we could have used `C` to unpack the individually accessible byte registers FL, FH, AL, AH, etc.:

```
my( $fl, $fh, $al, $ah, $bl, $bh, $cl, $ch, $dl, $dh ) =
  unpack( 'C10', substr( $frame, 4, 10 ) );
```

It would be nice if we could do this in one fell swoop: unpack a short, back up a little, and then unpack 2 bytes. Since Perl *is* nice, it proffers the template code `X` to back up one byte. Putting this all together, we may now write:

```
my( $ip, $cs,
    $flags,$fl,$fh,
    $ax,$al,$ah, $bx,$bl,$bh, $cx,$cl,$ch, $dx,$dl,$dh,
    $si, $di, $bp, $ds, $es ) =
unpack( 'v2' . ('vXXCC' x 5) . 'v5', $frame );
```

(The clumsy construction of the template can be avoided - just read on!)

We've taken some pains to construct the template so that it matches the contents of our frame buffer. Otherwise we'd either get undefined values, or `unpack` could not unpack all. If `pack` runs out of items, it will supply null strings (which are coerced into zeroes whenever the pack code says so).

32.4.3 How to Eat an Egg on a Net

The pack code for big-endian (high order byte at the lowest address) is `n` for 16 bit and `N` for 32 bit integers. You use these codes if you know that your data comes from a compliant architecture, but, surprisingly enough, you should also use these pack codes if you exchange binary data, across the network, with some system that you know next to nothing about. The simple reason is that this order has been chosen as the *network order*, and all standard-fearing programs ought to follow this convention. (This is, of course, a stern backing for one of the Lilliputian parties and may well influence the political development there.) So, if the protocol expects you to send a message by sending the length first, followed by just so many bytes, you could write:

```
my $buf = pack( 'N', length( $msg ) ) . $msg;
```

or even:

```
my $buf = pack( 'NA*', length( $msg ), $msg );
```

and pass `$buf` to your send routine. Some protocols demand that the count should include the length of the count itself: then just add 4 to the data length. (But make sure to read §32.7 before you really code this!)