

packet.lisp

Luke Gorrie

July 12, 2004

Contents

1	Introduction	1
2	Top-level interface	1
3	I/O machinery	2
3.1	Input "grabbing"	2
3.2	Output "shoving"	4
4	Protocol implementations	5
4.1	Ethernet	6
4.1.1	ethernet-address	6
4.1.2	Decode and encode	6
4.2	ARP	7
4.3	IPv4	8
4.3.1	ipv4-address	8
4.3.2	decoder	9
4.4	UDP	11
5	Checksum computation	12
6	Decoding driver	13
7	Encoding driver	14
8	Sample packets	14
9	Exporting structures	17

1 Introduction

This is a program for encoding and decoding the packet headers of some TCP/IP family protocols. It converts between packets represented as octet-vectors and structures.

This program is a library; it's not very useful in itself.

Written for CMUCL 19A. I've used some non-portable features: `ext:collect`, `slot-value` on structures, and PCL introspection.

```

;; Avoid calling defpackage is the package already exists.
;; Is this correct? It does avoid a lot of irritating warnings due to
;; the programmed exports at the end of the file.
(eval-when (:compile-toplevel :load-toplevel :execute)
  (unless (find-package "PACKET")
    (defpackage :packet
      (:use :common-lisp)
      ;; Note: structures and their accessors are auto-exported down below.
      (:export #:decode #:encode
               #:buffer #:octet #:packet #:header))))

(in-package :packet)

```

2 Top-level interface

The program has two main data types: buffers and packets. Buffers are octet-vectors for the encoded binary representations of packets.

```

(deftype buffer ()
  "A network packet represented as a vector of octets."
  '(array octet (*)))

```

```

(deftype octet ()
  "An unsigned 8-bit byte."
  '(unsigned-byte 8))

```

A packet is a list of header structures followed by zero or more buffers of raw data. This is the representation of a decoded network packet's headers and payload. Note that the individual header types are defined down below in the protocol-specific sections.

```

(deftype packet ()
  "A list of headers and buffers representing a network packet."
  'cons)

```

```

(deftype header ()
  "A decoded protocol header."
  '(or ethernet-header arp-header ipv4-header udp-header))

```

The decode and encode functions convert between the buffer and packet representations. They are inverse operations when applied to well-formed inputs.

```

(declare (ftype (function (buffer) packet) decode)
         (ftype (function (packet) buffer) encode))

```

```

(defun decode (buffer)
  "Decode BUFFER as a packet."
  (decode-headers buffer))

```

```

(defun encode (packet)
  "Encode PACKET into a buffer."
  (encode-headers packet))

```

3 I/O machinery

All our I/O is based on treating a `buffer` (octet-vector) as a stream of bits. For decoding we “grab” quantities of bits from the buffer as needed, and for encoding we similarly “shove” bits into an output buffer.

3.1 Input ”grabbing”

```
(defvar *decode-buffer* nil
  "Buffer containing the packet currently being decoded.")

(defvar *decode-position* nil
  "Current bit-position in *DECODE-BUFFER*.")

(defmacro with-buffer-input (buffer &body body)
  "Execute BODY, grabbing input from the beginning of BUFFER."
  `(let ((*decode-buffer* ,buffer)
        (*decode-position* 0))
    ,@body))

(defun bit-octet (bit &optional (check-alignment t))
  "Convert from bit position to octet position."
  (multiple-value-bind (quotient remainder) (truncate bit 8)
    (when (and check-alignment (plusp remainder))
      (error "Bit-position ~S is not octet-aligned." bit))
    quotient))

(defun octet-bit (octet)
  "Convert from octet position to bit position."
  (* 8 octet))

“Grab” functions consume input from *decode-buffer* and advance *decode-position*.

(defun grab-octets (num)
  "Grab a vector of NUM octets."
  (let ((start (bit-octet *decode-position*)))
    (incf *decode-position* (* num 8))
    (subseq *decode-buffer* start (+ num start))))

(defun grab-ethernet-address ()
  (make-ethernet-address :octets (grab-octets 6)))

(defun grab-ipv4-address ()
  (make-ipv4-address :octets (grab-octets 4)))

(defun grab-rest ()
  "Grab the rest of the buffer into an octet vector."
  (prog1 (subseq *decode-buffer* (bit-octet *decode-position*))
    (setf *decode-position* (octet-bit (length *decode-buffer*)))))
```

```
(defmacro dpb! (value bytespec place)
  "Deposit VALUE into BYTESPEC of PLACE."
  `(setf ,place (dpb ,value ,bytespec ,place)))
```

I've written this function countless times but it always comes out ugly. What's the right way?

```
(defun grab-bits (bits)
  "Grab a BITS-long unsigned integer"
  (let ((accumulator 0)
        (remaining bits))
    (flet ((accumulate-byte ()
            ;; Accumulate the relevant part of the current byte and
            ;; advance to the next one.
            (let* ((size (min remaining (- 8 (rem *decode-position* 8))))
                  (offset (rem (- 8 (rem (+ *decode-position* size) 8)) 8))
                  (value (ldb (byte size offset)
                              (aref *decode-buffer*
                                    (bit-octet *decode-position* nil)))))
              (decf remaining size)
              (dpb! value (byte size remaining) accumulator)
              (incf *decode-position* size))))
      (loop while (plusp remaining)
            do (accumulate-byte))
      accumulator)))
```

```
(defun grab-bitflag ()
  "Grab a single bit. Return T if it's 1 and NIL if it's 0."
  (= (grab-bits 1) 1))
```

3.2 Output "shoving"

```
(defvar *encode-buffer* nil
  "Buffer (adjustable and with fill-pointer) for encoding data into.")
```

```
(defvar *encode-position* nil
  "The encoding position in *ENCODE-BUFFER*.")
```

```
(defvar *encode-bit-bucket* 0 "Accumulator for smaller-than-byte output.")
(defvar *encode-bit-offset* 0 "The current accumulator bit-position.")
```

```
(defmacro with-buffer-output (() &body body)
  `(let ((*encode-buffer* (make-array '(0) :element-type 'octet
                                       :adjustable t :fill-pointer 0))
        (*encode-position* 0)
        (*encode-bit-bucket* 0)
        (*encode-bit-offset* 0))
    ,@body
    (coerce *encode-buffer* 'buffer)))
```

```
(defun encoding-position ()
```

```

(length *encode-buffer*))

(defmacro with-buffer-patch (position &body body)
  "Shove output at POSITION, overwriting any that was already there.
Used within WITH-BUFFER-OUTPUT."
  `(let ((*encode-position* ,position)
        ,@body))

"Shove" functions extend *encode-buffer* and advance *encode-position*.

(defun shove-octet (octet)
  (unless (zerop *encode-bit-offset*)
    (error "Attempt to shove an octet at unaligned position."))
  (cond ((= *encode-position* (length *encode-buffer*))
        (vector-push-extend octet *encode-buffer*)
        (incf *encode-position*))
        (< *encode-position* (length *encode-buffer*))
        (setf (aref *encode-buffer* *encode-position*) octet)
        (incf *encode-position*))
    (t
     (error "Can't shove to position ~D with ~D-length buffer!"
            *encode-position* (length *encode-buffer*)))))

(defun shove-ethernet-address (address)
  (shove-vector (ethernet-address.octets address)))

(defun shove-ipv4-address (address)
  (shove-vector (ipv4-address.octets address)))

(defun shove-vector (vector)
  (map nil #'shove-octet vector))

(defun shove-bits (value nbits)
  "Shove NBITS of VALUE."
  (cond ((zerop nbits)
        (< (+ *encode-bit-offset* nbits) 8)
        ;; We can fit NBITS into the accumulator without filling it.
        ;; Deposit VALUE into the most-significant accumulator bits
        ;; available.
        (let ((store-offset (- 8 nbits *encode-bit-offset*)))
          (dpp! value (byte nbits store-offset) *encode-bit-bucket*)
          (incf *encode-bit-offset* nbits)))
        (t
         ;; We have at least enough data to complete a byte. We
         ;; consume enough of VALUE's most-significant bits to fill
         ;; the accumulator, output a byte, then recurse on any
         ;; remainder.
         (let* ((take-bits (- 8 *encode-bit-offset*))
                ;; The TAKE-BITS most-significant bits of VALUE.
                (take-value (ldb (byte take-bits (- nbits take-bits)) value)))
           (shove-bits take-value (- nbits take-bits))))))

```

```

        (store-offset (- 8 take-bits *encode-bit-offset*))
        (remaining (- nbits take-bits)))
(dpb! take-value (byte take-bits store-offset) *encode-bit-bucket*)
(setf *encode-bit-offset* 0)
(shove-octet *encode-bit-bucket*)
;; Recurse with the remainder.
(shove-bits value remaining))))))

```

4 Protocol implementations

Each protocol defines a pair of GRAB and SHOVE functions for its headers.

```

(defvar *resolve-protocols* t
  "When non-nil protocol numbers are resolved to symbolic names.
Unrecognised numbers are left as numbers.")

(defun lookup (key alist &key (errorp t) (reversep nil))
  "Lookup the value of KEY in ALIST.
If the key is not present and ERRORP is true then an error is
signalled; if ERRORP is nil then the key itself is returned."
  (let ((entry (funcall (if reversep #'rassoc #'assoc) key alist)))
    (cond (entry (funcall (if reversep #'car #'cdr) entry))
          (errorp (error "Key ~S is not present in ~A." key alist))
          (t key))))))

(defun rlookup (key alist)
  "Lookup the value of KEY in CDR-position in ALIST, else return KEY."
  (lookup key alist :errorp nil :reversep t))

```

4.1 Ethernet

4.1.1 ethernet-address

This big `eval-when` is needed to define the read-syntax for `ethernet-address` such that it can be used in this file.

The read syntax is `#e"ff:00:1:2:3:4`.

```

(eval-when (:compile-toplevel :load-toplevel :execute)
  (defstruct (ethernet-address (:conc-name #:ethernet-address.))
    (:print-function print-ethernet-address))
  "48-bit Ethernet MAC address."
  (octets (ext:required-argument) :type (array octet (6))))

(defun read-ethernet-address (stream &optional c n)
  "Read an ethernet address in colon-separated syntax.
Example: #e\"1:2:3:4:5:6\""
  (declare (ignore c n))
  (let ((value-stream (make-string-input-stream (read stream t nil t)))
        (*readtable* (copy-readtable))
        (*read-base* 16))

```

```

(set-syntax-from-char #\: #\Space)
(let ((vec (make-array '(6) :element-type 'octet)))
  (dotimes (i 6)
    (let ((octet (read value-stream t nil t)))
      (unless *read-suppress*
        (setf (elt vec i) octet))))
    (unless *read-suppress*
      (make-ethernet-address :octets vec))))))

(set-dispatch-macro-character #\# #\e 'read-ethernet-address)

(defun print-ethernet-address (address stream depth)
  "Print ethernet addresses as in #e\"0:ff:1:2:3:4\"."
  (declare (ignore depth))
  (format stream "#e\"~{~16,2,'OR~^:~}\""
    (coerce (ethernet-address.octets address) 'list)))

(defmethod make-load-form ((s ethernet-address) &optional env)
  (make-load-form-saving-slots s :environment env))

```

4.1.2 Decode and encode

```

(defstruct (ethernet-header (:conc-name #:ethernet-header.))
  (dest nil :type (or null ethernet-address))
  (source nil :type (or null ethernet-address))
  (protocol nil :type (or null (unsigned-byte 16) symbol)))

(defparameter *ethernet-protocol-names* '((#x0806 . :arp) (#x0800 . :ipv4))
  "Mapping from ethernet protocol numbers to symbolic names.")

(defun grab-ethernet-header ()
  "Grab an ethernet header and call FUNCTION with each part."
  (let ((struct (make-ethernet-header)))
    (flet ((header (name value) (setf (slot-value struct name) value)))
      (header 'dest (grab-ethernet-address))
      (header 'source (grab-ethernet-address))
      (header 'protocol (ethernet-protocol-name (grab-bits 16))))
    struct))

(defun ethernet-protocol-name (number)
  "Return the symbolic protocol name of NUMBER, if appropriate."
  (if *resolve-protocols*
    (lookup number *ethernet-protocol-names* :errorp nil)
    number))

(defun shove-ethernet-header (header)
  (declare (type ethernet-header header))
  (with-slots (dest source protocol) header
    (shove-ethernet-address dest)
    (shove-ethernet-address source)))

```

```
(shove-bits (rlookup protocol *ethernet-protocol-names*) 16))
(constantly nil))
```

4.2 ARP

```
(defstruct (arp-header (:conc-name #:arp-header.))
  (hardware-type nil :type (or null (unsigned-byte 16)))
  (protocol-type nil :type (or null (unsigned-byte 16)))
  (hardware-length nil :type (or null (unsigned-byte 8)))
  (protocol-length nil :type (or null (unsigned-byte 8)))
  (operation nil :type (or null symbol (unsigned-byte 16)))
  (sender-ha nil :type (or null ethernet-address))
  (sender-ip nil :type (or null ipv4-address))
  (target-ha nil :type (or null ethernet-address))
  (target-ip nil :type (or null ipv4-address)))

(defun grab-arp-header ()
  "Grab an ARP header and call FUNCTION with each part."
  (let ((struct (make-arp-header)))
    (flet ((header (name value) (setf (slot-value struct name) value)))
      (header 'hardware-type (grab-bits 16))
      (header 'protocol-type (grab-bits 16))
      (header 'hardware-length (grab-bits 8))
      (header 'protocol-length (grab-bits 8))
      (header 'operation (arp-operation (grab-bits 16)))
      (header 'sender-ha (grab-ethernet-address))
      (header 'sender-ip (grab-ipv4-address))
      (header 'target-ha (grab-ethernet-address))
      (header 'target-ip (grab-ipv4-address)))
    struct))

(defvar *arp-operation-names* '((1 . :request) (2 . :response))
  "Mapping between ARP operation numbers and their symbolic names.")

(defun arp-operation (operation)
  "Return the symbolic name for OPERATION, if appropriate."
  (if *resolve-protocols*
      (lookup operation *arp-operation-names* :errorp nil)
      operation))

(defun shove-arp-header (header)
  (declare (type arp-header header))
  (with-slots (hardware-type protocol-type hardware-length protocol-length
              operation sender-ha sender-ip target-ha target-ip)
    header
    (shove-bits hardware-type 16)
    (shove-bits protocol-type 16)
    (shove-octet hardware-length)
    (shove-octet protocol-length)
    (shove-bits (rlookup operation *arp-operation-names*) 16)
```

```

(shove-ethernet-address sender-ha)
(shove-ipv4-address sender-ip)
(shove-ethernet-address target-ha)
(shove-ipv4-address target-ip))
(constantly nil))

```

4.3 IPv4

The Internet Protocol is described in RFC791.

4.3.1 ipv4-address

IP addresses also have a special read-syntax: @10.0.0.1.

```

(eval-when (:compile-toplevel :load-toplevel :execute)
  (defstruct (ipv4-address (:conc-name #:ipv4-address.)
                (:print-function print-ipv4-address))
    (octets (ext:required-argument) :type (array octet (4))))

  (defun read-ipv4-address (stream &optional c n)
    "Read an IPv4 address in dotted-quad format.
    Example: @192.168.0.1"
    (declare (ignore c n))
    (let ((*readtable* (copy-readtable)))
      (set-syntax-from-char #\. #\Space)
      (let ((vec (make-array '(4) :element-type 'octet)))
        (dotimes (i 4)
          (let ((octet (read stream t nil t)))
            (unless *read-suppress*
              (setf (elt vec i) octet))))))
        (unless *read-suppress*
          (make-ipv4-address :octets vec))))))

  (set-macro-character #\@ 'read-ipv4-address t)

  (defun print-ipv4-address (address stream depth)
    "Print IPv4 addresses as in @192.168.0.1."
    (declare (ignore depth))
    (format stream "@~{~A~^.~}" (coerce (ipv4-address.octets address) 'list)))

  (defmethod make-load-form ((s ipv4-address) &optional env)
    (make-load-form-saving-slots s :environment env)))

```

4.3.2 decoder

```

(defstruct (ipv4-header (:conc-name #:ipv4-header.))
  (version      nil :type (or null (unsigned-byte 4)))
  (hlen         nil :type (or null (unsigned-byte 4)))
  (tos          nil :type (or null (unsigned-byte 8)))
  (total-len   nil :type (or null (unsigned-byte 16)))
  (id          nil :type (or null (unsigned-byte 16)))

```

```

(flags          nil :type (or null (unsigned-byte 3)))
(fragment-offset nil :type (or null (unsigned-byte 13)))
(ttl           nil :type (or null (unsigned-byte 8)))
(protocol      nil :type (or null symbol (unsigned-byte 8)))
(checksum      nil :type (or null (unsigned-byte 16)))
(source        nil :type (or null ipv4-address))
(dest         nil :type (or null ipv4-address))

(defconstant ipv4-min-hlen 5
  "The header length (in 32-bit words) of an IPv4 packet with no options.")

(defun grab-ipv4-header ()
  (let ((struct (make-ipv4-header))
        (header-start-pos (bit-octet *decode-position*))
        hlen
        checksum)
    (flet ((header (name value) (setf (slot-value struct name) value)))
      (header 'version      (grab-bits 4))
      (header 'hlen        (setf hlen (grab-bits 4)))
      (header 'tos         (grab-bits 8))
      (header 'total-len   (grab-bits 16))
      (header 'id          (grab-bits 16))
      (header 'flags       (grab-bits 3))
      (header 'fragment-offset (grab-bits 13))
      (header 'ttl         (grab-bits 8))
      (header 'protocol    (if *resolve-protocols*
                              (ipv4-protocol (grab-bits 8))
                              (grab-bits 8)))
      (header 'checksum    (setf checksum (grab-bits 16)))
      (header 'source      (grab-ipv4-address))
      (header 'dest        (grab-ipv4-address))
      ;; FIXME
      (unless (= hlen ipv4-min-hlen)
        (error "Can't decode options in IPv4 packets."))
      (let* ((initial (- checksum))
             (header-octets (* hlen 4))
             (computed-checksum
              (checksum *decode-buffer*
                       :position header-start-pos
                       :length header-octets
                       :initial initial)))
        (unless (eql checksum computed-checksum)
          (error "Bad checksum: Got ~D, computed ~D."
                 checksum computed-checksum))))
      struct))

(defvar ipv4-protocol-names '((1 . :icmp) (6 . :tcp) (17 . :udp))
  "Mapping between IPv4 protocol numbers and their symbolic names.")

(defun ipv4-protocol (number)

```

```

"Return the symbolic name for protocol NUMBER, if appropriate."
(if *resolve-protocols*
    (lookup number ipv4-protocol-names :errorp nil)
    number))

(defconstant ipv4-no-options-hlen 5)

(defun shove-ipv4-header (header)
  "Shove an IPv4 header.
The length and checksum fields are computed automatically."
  (with-slots (version hlen tos total-len id flags fragment-offset ttl
              protocol checksum source dest)
    header
    (let (start total-len-pos checksum-pos)
      (setf start (encoding-position))
      (shove-bits version 4)
      (shove-bits ipv4-no-options-hlen 4)
      (shove-bits tos 8)
      (setf total-len-pos (encoding-position))
      (shove-bits 0 16) ; total-len
      (shove-bits id 16)
      (shove-bits flags 3)
      (shove-bits fragment-offset 13)
      (shove-bits ttl 8)
      (shove-bits (rlookup protocol ipv4-protocol-names) 8)
      ;; Remember where the checksum is: we have to come back and
      ;; patch it in.
      (setf checksum-pos (encoding-position))
      (shove-bits 0 16)
      (shove-ipv4-address source)
      (shove-ipv4-address dest)
      (lambda ()
        (with-buffer-patch total-len-pos
          (let ((total-len (- (length *encode-buffer*) start)))
            (shove-bits total-len 16)))
        (with-buffer-patch checksum-pos
          (shove-bits (checksum *encode-buffer*
                               :position start
                               :length (* ipv4-no-options-hlen 4))
                     16))))))

```

4.4 UDP

RFC 768

```

(defstruct (udp-header (:conc-name #:udp-header.))
  (src-port nil :type (or null (unsigned-byte 16)))
  (dest-port nil :type (or null (unsigned-byte 16)))
  (length nil :type (or null (unsigned-byte 16)))
  (checksum nil :type (or null (unsigned-byte 16))))

```



```

                :position start-pos
                :initial (udp-pseudo-header-checksum-acc
                          src-ip dest-ip length)))
    (with-buffer-patch checksum-pos
      (shove-bits csum 16))))))

```

5 Checksum computation

The TCP/IP protocols use 16-bit ones-complement checksums. See RFC1071 for details.

```

(defun checksum (buffer
                &key
                (position 0)
                (length (- (length buffer) position))
                (initial 0)
                (finish t))
  "Compute a checksum using normal twos-complement arithmetic.
The buffer is treated as a sequence of 16-bit big-endian numbers."
  (declare (type buffer buffer))
  (let ((last-pos (+ position (1- length)))
        (acc initial))
    (do ((msb-pos position (+ msb-pos 2))
        (lsb-pos (1+ position) (+ lsb-pos 2)))
        ((> msb-pos last-pos))
      (let ((msb (aref buffer msb-pos))
            (lsb (if (> lsb-pos last-pos) 0 (aref buffer lsb-pos))))
        (incf acc (dpb msb (byte 8 8) lsb))))
      (if finish (finish-checksum acc) acc)))

(defun finish-checksum (n)
  "Convert N into an unsigned 16-bit ones-complement number.
The result is a bit-pattern also represented as an integer."
  (let* ((acc (+ (ldb (byte 16 16) n)
                 (ldb (byte 16 0) n)))
         (acc (+ acc (ldb (byte 16 16) acc))))
    (logxor #xFFFF (ldb (byte 16 0) acc))))

(defun checksum-acc-ipv4-address (address)
  "Return the partial checksum accumulated from an IPv4 address."
  (checksum (ipv4-address.octets address) :finish nil))

```

6 Decoding driver

```

(defvar *previous-header* nil
  "Bound to the previously decoded header.
Some protocols (e.g. UDP) need to retrieve fields from their enclosing
protocol's header.")

```

```

(defun decode-headers (buffer)
  "Return a list of decoded headers from BUFFER"
  (with-buffer-input buffer
    (let* ((headers (grab-more-headers (grab-header :ethernet)))
           (rest (grab-rest)))
      (if (zerop (length rest))
          headers
          (append headers (list rest))))))

(defun grab-more-headers (header)
  "Accumulate HEADER and continue decoding the rest."
  (if (member (type-of header) '(ethernet-header ipv4-header))
      (let ((*previous-header* header)
            (inner-protocol (slot-value header 'protocol)))
        (cons header (grab-more-headers (grab-header inner-protocol))))
      ;; This is the last header we know how to decode.
      (list header)))

(defun grab-header (protocol)
  "Grab and return the header of PROTOCOL."
  (case protocol
    (:ethernet (grab-ethernet-header))
    (:arp (grab-arp-header))
    (:ipv4 (grab-ipv4-header))
    (:udp (let ((src-ip (slot-value *previous-header* 'source))
                 (dest-ip (slot-value *previous-header* 'dest)))
            (grab-udp-header src-ip dest-ip))))

```

7 Encoding driver

To encode a packet we “shove” each element into a vector and then apply the “touchup functions”.

```

(defun encode-headers (headers)
  (let (src-ip dest-ip)
    (flet ((shove-element (e)
            ;; Shove E into the encoding vector and return a touchup.
            (etypecase e
              (buffer
               (shove-vector e)
               (constantly nil))
              (ethernet-header
               (shove-ethernet-header e))
              (arp-header
               (shove-arp-header e))
              (ipv4-header
               (setf src-ip (ipv4-header.source e))
               (setf dest-ip (ipv4-header.dest e))
               (shove-ipv4-header e))

```

```

        (udp-header
         (assert (and src-ip dest-ip))
         (shove-udp-header e src-ip dest-ip))))
(with-buffer-output ()
 (let ((touchups '()))
  (dolist (e headers)
   (push (shove-element e) touchups))
  (mapc #'funcall touchups))))

```

8 Sample packets

```

(defparameter *arp-packet*
  (coerce
   #(255 255 255 255 255 255 0 8 116 228 110 188 8 6 0 1 8 0 6 4 0 1 0 8 116
     228 110 188 192 168 128 44 0 0 0 0 0 0 192 168 128 1)
   'buffer)
  "An ethernet frame containing an ARP request.")

(defparameter *udp-packet*
  (coerce
   #(255 255 255 255 255 255 0 8 116 228 110 188 8 0 69 0 0 124 0 0 64 0 64
     17 183 244 192 168 128 44 192 168 128 255 128 117 0 111 0 104 5 206 20
     15 249 61 0 0 0 0 0 0 2 0 1 134 160 0 0 0 2 0 0 0 5 0 0 0 1 0 0 0 24
     64 158 126 39 0 0 0 4 100 111 100 111 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
     0 0 0 0 1 134 164 0 0 0 2 0 0 0 2 0 0 0 16 0 0 0 12 98 108 117 101 116
     97 105 108 46 99 111 109)
   'buffer)
  "An ethernet frame containing a UDP packet.")

(defun test ()
  (decode-test)
  (encode-test)
  'ok)

(defun decode-test ()
  "Test that the sample packets are decoded correctly."
  (let* ((arp-headers (decode *arp-packet*))
         (udp-headers (decode *udp-packet*)))
    (check-headers arp-headers
      '((ETHERNET-HEADER
        ((DEST . #e"FF:FF:FF:FF:FF:FF")
         (SOURCE . #e"00:08:74:E4:6E:BC")
         (PROTOCOL . :ARP)))
      (ARP-HEADER
        ((HARDWARE-TYPE . 1)
         (PROTOCOL-TYPE . 2048)
         (HARDWARE-LENGTH . 6)
         (PROTOCOL-LENGTH . 4)
         (OPERATION . :REQUEST))

```

```

        (SENDER-HA      . #e"00:08:74:E4:6E:BC")
        (SENDER-IP      . @192.168.128.44)
        (TARGET-HA      . #e"00:00:00:00:00:00")
        (TARGET-IP      . @192.168.128.1))))))
(check-headers udp-headers
 '( (ETHERNET-HEADER
    (DEST      . #e"FF:FF:FF:FF:FF:FF")
    (SOURCE    . #e"00:08:74:E4:6E:BC")
    (PROTOCOL  . :IPV4)))
  (IPV4-HEADER
    (VERSION   . 4)
    (HLEN      . 5)
    (TOS       . 0)
    (TOTAL-LEN . 124)
    (ID        . 0)
    (FLAGS     . 2)
    (FRAGMENT-OFFSET . 0)
    (TTL       . 64)
    (PROTOCOL  . :UDP)
    (CHECKSUM  . 47092)
    (SOURCE    . @192.168.128.44)
    (DEST      . @192.168.128.255)))
  (UDP-HEADER
    (SRC-PORT  . 32885)
    (DEST-PORT . 111)
    (LENGTH    . 104)
    (CHECKSUM  . 1486)))
, (coerce
  #(20 15 249 61 0 0 0 0 0 0 2 0 1 134 160 0 0 0
    2 0 0 0 5 0 0 0 1 0 0 0 24 64 158 126 39 0 0 0
    4 100 111 100 111 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
    0 0 0 0 0 0 1 134 164 0 0 0 2 0 0 0 2 0 0 0 16 0
    0 0 12 98 108 117 101 116 97 105 108 46 99 111 109)
  'buffer))))))

(defun check-headers (headers specs)
  "Check that HEADERS agrees element-wise with SPECS.
SPECS is a list of specifications of what a header should contain."
  (flet ((check (header spec)
    ;; Raise an error if HEADER doesn't match SPEC.
    (loop for (slot . value) in (second spec)
      do (unless (equalp (slot-value header slot) value)
        (error "Slot ~A: Expected ~A, got ~A."
              slot value (slot-value header slot))))
    always t)))
  (unless (and (null headers) (null specs))
    (let ((header (first headers))
          (spec (first specs)))
      (if (and (typep header 'buffer) (typep spec 'buffer))
          (unless (equalp header spec)
            (error "Slot ~A: Expected ~A, got ~A."
                  (slot-value header (first spec))
                  (slot-value spec (first spec))
                  (first spec))))
          (unless (equalp header spec)
            (error "Slot ~A: Expected ~A, got ~A."
                  (slot-value header (first spec))
                  (slot-value spec (first spec))
                  (first spec)))))))

```

```

        (error "Mismatch in binary parts.))
      (progn
        (unless (eq (type-of header) (first spec))
          (error "Header type mismatch: ~A ~A"
                 (type-of header) (first spec)))
        (check header spec)
        (check-headers (rest headers) (rest specs)))))))))

(defun encode-test ()
  "Check that (encode (decode PACKET)) <=> identity."
  (assert (and (equalp *udp-packet* (encode (decode *udp-packet*)))
               (equalp *arp-packet* (encode (decode *arp-packet*))))))

(defun bench (n)
  "Show how long it takes to decode and re-encode 10^N UDP packets."
  (time (dotimes (i (expt 10 n))
                (encode (decode *udp-packet*)))))

```

9 Exporting structures

My pet hate is explicitly enumerating all the accessors for structures in export declarations. Instead we do a little introspection to enumerate them automatically, and jump through some hoops with `defpackage` (above) to avoid warnings.

```

(eval-when (:compile-toplevel :load-toplevel)

  (defun structure-exports ()
    "The list of defstruct-defined symbols that we want to export."
    (apply #'append (mapcar #'structure-symbol-names
                            '(ethernet-header ethernet-address
                              arp-header
                              ipv4-header ipv4-address
                              udp-header))))

  (defun structure-symbol-names (name)
    "Return all the interesting symbols generated by DEFSTRUCT for NAME.
    Assumes a FOO type name, MAKE-FOO constructor, and FOO-P
    predicate to avoid excessively low-level introspection."
    (list* name
           (find-symbol (format nil "~A-P" name))
           (find-symbol (format nil "MAKE-~A" name))
           (structure-accessors name)))

  (defun structure-accessors (name)
    (mapcar #'pcl::slot-definition-defstruct-accessor-symbol
            (pcl:class-direct-slots (find-class name))))

  (export (structure-exports)))

```