```
L           JJJ      SSSS
L            J      S   S
L            J      S
L            J       SSS
L            J          S
L   L     J  J      S   S
LLLLL      JJJ      SSSS
```

Wed 30-Apr-1986 12:51:25

Print request number 143

Station: $36

Name: L J Shustek

File Server: BEETHOVEN ($F2)

NFS Pathname:

Filename (s):

Print Server: LENNON ($8A)
Printer: LASER
Setup: LANDSCAPE
Priority: Standard
Copies: 1
Eject: 0

```
    @@@@    @@@@@@@@@  @@@@@@@@@@@@@@@@@@@@@@@@      @@@@@@@
    @@@@    @@@@@@@@  @@@@@@@@@@@@@@@@@@@@@@@@@@     @@@@@@@@
   @@@@@@  @@@       @@@@          @@@    @@@@@@    @@@   @@@@
   @@@@@@  @@@        @@@         @@@@   @@@ @@@    @@@   @@@@
   @@@@@@ @@@@@@@@@   @@@@@@      @@@   @@@   @@@   @@@@@@@@@
  @@@  @@@@@@@@@@@      @@@@@@   @@@@  @@@@@@@@@@  @@@@@@@@@
  @@@  @@@@@               @@@   @@@  @@@@@@@@@@@  @@@  @@@
 @@@@  @@@@@               @@@@  @@@ @@@      @@@@ @@@   @@@
 @@@    @@@@@@@@@@@@@@@@@@@@@@  @@@ @@@        @@@@@@    @@@
@@@@    @@@@@@@@@@@@@@@@@@@@@   @@@@@@         @@@@@@    @@@
```

```
/* this is l4.h */


/*
                 ******************************************
                 *                                        *
                 *    Fileserver prototype - TRANSPORT    *
                 *    --------------------------------    *
                 *                                        *
                 *           Module "l4.h"                *
                 *                                        *
                 ******************************************


This file contains the definitions of symbols and structures used
by callers and implementation modules of the Network Transport
(Level 4) routines.  These routines implement XNS transport protocol
for both Arcnet and Token Ring network.

*/




/*------------------------------------------------------------------------------


      Transport Level status codes


    These are the values for the "status" field of the L4 request
    block, and indicate the condition of the connection and/or the
    most recent request.

------------------------------------------------------------------------------*/

#define     l4st_uncon     0    /* unconnected */
#define     l4st_busy      1    /* connected, and command is in progress */
#define     l4st_done      2    /* connected, and command executed ok */



#define      RBid    0x5242    /* 'RB' for rb->id field */
```

```
/*---------------------------------------------------------------------------


    Packet Header


    This defines the format of incoming packets.
    All fields except the host source and destination addresses
    (srchost and dsthost) and source and destination sockets
    are private to the L4 routines.

    All multibyte fields are stored MSB first.

---------------------------------------------------------------------------*/


struct pkthdr {             /* packet header: datalink + Nestar + XNS */

    byte        sysid;          /* Datapoint-administered system id */
    byte        garbage;        /* garbage count */
    byte        pktno;          /* fragmentation packet number */
    byte        maxpkt;         /* max pkt size (bits 7..6), and
                                   fragmentation fragment number (bit 5..1) */
    word        chksum;         /* checksum (start of XNS packet) */
    word        length;         /* length, starting from checksum */
    byte        trnctl;         /* transport control byte */
    byte        ptype;          /* internet packet type (5 for SPP) */
    lword       dstnw;          /* destination network */
    uword       dsthost[3];     /* destination host    PUBLIC VARIABLE */
    uword       dstskt;         /* destination socket  PUBLIC VARIABLE */
    lword     . srcnw;          /* source network */
    uword       srchost[3];     /* source host         PUBLIC VARIABLE */
    uword       srcskt;         /* source socket       PUBLIC VARIABLE */
    byte        conctl;         /* connection control byte */
    byte        dtype;          /* datastream type */
    uword       srcid;          /* source connection id */
    uword       dstid;          /* destination connection id */
    uword       seqno;          /* sequence number */
    uword       ackno;          /* acknowledge number */
    uword       allno;          /* allocation number */

    };
```

```
/*--------------------------------------------------------------------------


    Transport Level Request Block


    This is the shared data structure which is used for parameters
    and state information for a connection.  It is allocated by
    the caller and passed to all L4 routines which are connection-
    specific.

    The first part are public variables which are set or examined by
    the caller, as indicated in the individual routine descriptions.

    The second part is a copy of the packet header and is "semi-public"
    in that only the host source and destination addresses are to be
    used by the caller;  other fields are private to the implementation.

    The third part contains private variables for use only by the
    implementation routines.

--------------------------------------------------------------------------*/


struct l4rb {


    /*  Public part  */

    uword            id;           /* id field 'RB' */
    struct l4rb     *flink;        /* forward link field for queues and lists */
    struct l4rb     *blink;        /* backward link field for queues and lists */
    addr             user;         /* arbitrary "user" field */
    void           (*anr)();       /* Asynchronous Notification Routine */
    int              status;       /* one of the l4st_xxx values */
    addr             rcvptr;       /* receive buffer address */
    word             rcvlength;    /* receive data length */
    word             rcvlimit;     /* receive buffer size */
    addr             sndptr;       /* send buffer address */
    word             sndlength;    /* send buffer length */
    byte             sndtype;      /* send datastream type */
    byte             rcvtype;      /* receive datastream type */
    uword            wks;          /* well-known socket to send on */
    boolean          arcnet;       /* is this an arcnet station? */


    /*  Semi-public part  */

    struct pkthdr    ph;           /* our transmit packet header */


    /*  Private part  */

    short int       state;        /* internal state */
    struct l4rb     *conlink;     /* link field connection list */
    short int       snd_count;    /* count of send retries */
    short int       timer;        /* countdown for pkt wait timeout */
    addr            bufcursor;    /* next position in the send or rcv buffer */
    short int       bytes_left;   /* # bytes left to send */
        short int               first_seq;   /* sequence number of 1st pkt of outgoing msg */
    boolean         on_a_list;    /* we are on a waiting list */
    boolean         we_owe_ack;   /* we owe him an ack */
    boolean         send_ack;     /* send an ack if 1-packet message */
    );
```

```
/* end of 14.h */
```

```
L            JJJ      SSSS
L             J      S    S
L             J      S
L             J       SSS
L             J           S
L    L      J J      S    S
LLLLL        JJJ     SSSS
```

Wed 30-Apr-1986 12:51:06

Print request number 142

Station: $36

Name: L J Shustek

File Server: BEETHOVEN ($F2)

NFS Pathname:

Filename (s):

```
Print Server: LENNON ($8A)
     Printer: LASER
       Setup: LANDSCAPE
    Priority: Standard
      Copies: 1
       Eject: 0
```

```
    @@@@    @@@@@@@@@  @@@@@@@@@@@@@@@@@@@@@@@@@     @@@@@@@
    @@@@    @@@@@@@@  @@@@@@@@@@@@@@@@@@@@@@@@@@@    @@@@@@@@
   @@@@@@  @@@       @@@@          @@@    @@@@@@    @@@   @@@@
   @@@@@@  @@@        @@@         @@@@   @@@ @@@    @@@   @@@@
   @@@@@@ @@@@@@@@@   @@@@@@      @@@   @@@   @@@   @@@@@@@@@
  @@@  @@@@@@@@@@@      @@@@@@   @@@@  @@@@@@@@@@  @@@@@@@@@
  @@@  @@@@@               @@@   @@@  @@@@@@@@@@@  @@@  @@@
 @@@@  @@@@@               @@@@  @@@ @@@      @@@@ @@@   @@@
 @@@    @@@@@@@@@@@@@@@@@@@@@@  @@@ @@@        @@@@@@    @@@
@@@@    @@@@@@@@@@@@@@@@@@@@@   @@@@@@         @@@@@@    @@@
```

```
/* this is l4private.h */


/*
               ******************************************
               *                                        *
               *    Fileserver prototype - TRANSPORT    *
               *    ********************************    *
               *                                        *
               *         Module "l4private.h"           *
               *                                        *
               ******************************************


This file contains private symbols and declarations for the transport
implementation.


/*------------------------------------------------------------------------------

                            Macros

------------------------------------------------------------------------------*/


/* Make a trace table entry */

/* Note that l4_trace calls provide two information fields, both of
   which were originally use for debugging.  The current macro only
   uses the first field and so is compatible with the trace routines
   used in the rest of the server.
*/

#define  l4_trace(c,i1,i2)  {do_trace(c,(lword)(i1));}


/* Hash a 6-byte station address into a connnection table index.

   The argument is the address of the 6-byte XNS address.
   The value of the macro is a number in the range 0..HASH_TABLE_SIZE
   which is the index into "l4_con_table" of a chain of rbs
   with open connections.

   (The current hash function uses the low byte of the XNS address,
    which is the Arcnet id and is hence optimmal for Arcnet.  In the
    usual case of one connection per pair of stations, there will be
    only one rb on the hash list.)
*/

#define  HASH_TABLE_SIZE 256

#if intel
#define  hash_addr(p)  ((p)[2] >>8)
#else
#define  hash_addr(p)  ((p)[2] & 0xff)
#endif

/*------------------------------------------------------------------------------

                     Miscellaneous symbols
```

```
-------------------------------------------------------------------------------*/

#define MAX_SOCKETS    2       /* max # of well-known sockets we can listen on */
#define NESTAR_SYSID   0xfe    /* datapoint-assigned system id for nestar protocol */
#define MAX_WKS        2999    /* the biggest well-known socket number */
#define MAX_XNS_PKT    508     /* the largest xns packet */
```

```
/*---------------------------------------------------------------------------

                        Timeout values

    These are in "ticks" which correspond to the frequency at which the
    l2_timerint routine is called.  Something like 200 milliseconds seems right.

    Note that no timeout value should be less than 2, since the actual value
    will vary between one less than the timeout and the timeout itself.

---------------------------------------------------------------------------*/

#define  TO_AWAIT_MSG   25    /*  5 sec: rcv msg wait time.  (Could be infinite.) */

#define  TO_AWAIT_PKT    5    /*  1 sec: intra-message data packet wait time */

#define  TO_AWAIT_ACK    4    /* .8 sec: ack wait time */

#define  TO_XMIT_PKT     3    /* .6 sec: transmit packet timeout */
                              /* (Implemented in Level 2;  change it there!) */

#define  TO_PKT_DISCARD  2    /* .4 sec: unclaimed packet discard time */
                              /* (Should be minimal; set higher for debug) */


/*  Setting timeouts for maximum error recovery is a something of a black art.
    One set of relationships that seems to make sense is:

    TO_PKT_DISCARD  <  TO_XMIT_PKT  <  TO_AWAIT_ACK  <  TO_AWAIT_PKT

    This ordering attempts to encourage the following behavior:

    (1) Unclaimed received packets won't hang around long enough to
    prevent a transmission because there are no receive buffers available.

    (2) Receive timeouts won't occur until transmissions are unblocked.

    (3) A lost data packet will cause the sending to timeout waiting for
    the ack and thus retransmit before the receiver times out waiting for
    the missing data packet.


    There are still some circumstances where error recovery will fail,
    particularly near the beginning or end of a connection.  Example:  If the
    final ACK for a connection seems to the transmitter to have been sent,
    but the receiver didn't get it, the transmitter will close the RB and
    the repeated last message from the receiver will be discarded.  The
    receiver will then abort the connection.  So it goes.

    Note that having the Arcnet Level 2 retransmit if it gets TA without
    TMA solves that particular problem, but not all datalink hardware is as
    good as Arcnet.
*/

/*---------------------------------------------------------------------------

                    Maximum retry counts

---------------------------------------------------------------------------*/


#define  SEND_RETRIES   2       /* number of message send retry attempts to
```

```
                                make because the ACK wasn't received. */

#define  XMIT_RETRIES   1       /* number of packet send retry attempts to make
                                   if Level 2 detects an error. */
                                /* (Implemented in Level 2;  change it there!) */
```

```
/*---------------------------------------------------------------------------

            Internal 14 states in the rb->state field

---------------------------------------------------------------------------*/

#define  st_idle        0        /* idle;  no command in progress */

#define  st_connectg    1        /* starting a connect message (maybe on waitbuf list) */
#define  st_sending     2        /* sending a message          (maybe on waitbuf list) */
#define  st_sendackr    3        /* sending an ack (ack_req)   (maybe on waitbuf list) */
#define  st_sendackc    4        /* sending an ack (connected) (maybe on waitbuf list) */
#define  st_sendackd    5        /* sending an ack (disconn)   (maybe on waitbuf list) */

#define  st_pktwait     6        /* awaiting a message packet  (maybe on waitpkt list) */
#define  st_ackwait     7        /* awaiting an ack            (maybe on waitpkt list) */

#define  st_connwait    8        /* awaiting a connection      (on waitcon list) */



/*---------------------------------------------------------------------------

          XNS connection control bits

---------------------------------------------------------------------------*/


#define  syspkt  0x80       /* system packet (no data) */
#define  ackreq  0x40       /* ack request */
#define  attn    0x20       /* attn request */
#define  eom     0x10       /* end-of-message */
```

```
/*--------------------------------------------------------------------------------

        The format of ARC packets

--------------------------------------------------------------------------------*/


struct arc_header {   /* the arcnet packet header */

    byte  sid;        /* arcnet source address */
    byte  did;        /* arcnet destination address */
    byte  count1;     /* short packet data pointer, 0 if long packet */
    byte  count2;     /* long packet data pointer */

    /*  Thence follows empty space. */

    /*  Thence follows the data, bottom justified in the buffer to either
        256 bytes (short packet), or 512 bytes (long packet).  */


    };

#define MAXPKT_LONG 0x40    /* bit in ph.maxpkt that says long packets are ok */


/*--------------------------------------------------------------------------------

    Far pointer declarations for Microsoft C on the PC

(For the 68k implementation, these are the same as any other pointers.)

--------------------------------------------------------------------------------*/

#if microsoft
#define FARPTR far
#else
#define FARPTR
#endif

typedef char               FARPTR  *faraddr;    /* long address of char */
typedef struct pkthdr      FARPTR  *farphaddr;  /* long address of xns header */
typedef struct arc_header FARPTR  *fararcaddr; /* long address of arc_header */
```

```
/*-----------------------------------------------------------------------------

        Private static variables for L4

-----------------------------------------------------------------------------*/


short int  sockets [MAX_SOCKETS];   /* The array of valid sockets. */

boolean l4_busy = FALSE;            /* "We are in L4."  Used to prevent ANR recursion. */

short int  src_conid = 1;           /* The next source connnection id to use. */

short int  eph_socket = MAX_WKS+1;  /* The next ephemeral socket to use. */

word  our_addr[3];                  /* Our 48-bit network address. */


/*
        Variables which keep track of the RBs we know about.
*/

/*   The connection table (l4_con_table) is a hash table of pointers to RBs
     that have open connections.  The hash function is the macro hash_addr.
     RBs are chained from the hash table entry by the link field "conlink"
     from the time the connection is established until it is broken.
 */

struct l4rb *l4_con_table [HASH_TABLE_SIZE];   /* connection hash table */


/*   The RB waiting lists are used only when the RB has an outstanding
     command being processed by Level 4.
*/

struct l4_list {                    /* RB lists.                               */
   struct l4rb *head, *tail }       /*    Single threaded with ptrs to head and tail */

      waitbuf_list  = {NIL, NIL},       /* rbs waiting for a transmit buffer   */
      waitpkt_list  = {NIL, NIL},       /* rbs waiting for incoming packets    */
      waitcon_list  = {NIL, NIL};       /* rbs waiting for incoming connection */



/*
        Variables which control the disposition of the current incoming packet.

*/


farphaddr    rcvbuf_xptr  = NIL;  /* pointer to XNS header part of received packet */
fararcaddr   rcvbuf_aptr  = NIL;  /* pointer to arc header part of received packet */
short int    rcvbuf_timer = 0;    /* age of the received packet */
struct l4rb *rcvbuf_rb    = NIL;  /* the rb which owns the packet, if any */



/*
        Variables which record that various interrupts are pending.
*/

boolean  intpending_rcv   = FALSE;  /* receive interrupt pending */
boolean  intpending_xmit  = FALSE;  /* transmit buffer interrupt pending */
boolean  intpending_timer = FALSE;  /* timer interrrupt pending */
```

```
fararcaddr intpending_bufp;        /* argument for pending l2_rcvintr() */
faraddr    intpending_outbuf;      /* argument for pending l2_gotbuf() */

/* end of l4private.h */
```

```
L            JJJ      SSSS
L              J     S    S
L              J     S
L              J      SSS
L              J          S
L    L     J   J     S    S
LLLLL       JJJ      SSSS
```

Wed 30-Apr-1986 12:51:34

Print request number 144

Station: $36

Name: L J Shustek

File Server: BEETHOVEN ($F2)

NFS Pathname:

Filename (s):

Print Server: LENNON ($8A)
Printer: LASER
Setup: LANDSCAPE
Priority: Standard
Copies: 1
Eject: 0

```
    @@@@     @@@@@@@@@  @@@@@@@@@@@@@@@@@@@@@@@@       @@@@@@@
    @@@@     @@@@@@@@  @@@@@@@@@@@@@@@@@@@@@@@@@@      @@@@@@@@
   @@@@@@   @@@        @@@@          @@@     @@@@@@     @@@    @@@@
   @@@@@@   @@@        @@@          @@@@    @@@ @@@     @@@    @@@@
   @@@@@@ @@@@@@@@@   @@@@@@        @@@    @@@    @@@    @@@@@@@@@
  @@@   @@@@@@@@@@@      @@@@@@    @@@@   @@@@@@@@@@   @@@@@@@@@
  @@@   @@@@@           @@@    @@@    @@@@@@@@@@@   @@@   @@@
 @@@@   @@@@@           @@@@   @@@ @@@       @@@@ @@@     @@@
 @@@     @@@@@@@@@@@@@@@@@@@@@@  @@@ @@@          @@@@@@     @@@
@@@@     @@@@@@@@@@@@@@@@@@@@@   @@@@@@           @@@@@@     @@@
```

```
/* this is l4counts.h */


/*
                ******************************************
                *                                        *
                *    Fileserver prototype - TRANSPORT    *
                *    *********************************   *
                *                                        *
                *          Module "l4counts.h"           *
                *                                        *
                ******************************************


This file contains the declarations of the event counters for Level 4.

If the symbol L4GLOBALS is defined, this allocates storage for the
counters.  Otherwise it generates external references to the counters.

*/

#ifndef L4GLOBALS
#define counter(name) extern unsigned int name
#else
#define counter(name) unsigned int name = 0
#endif

counter (l4cnt_connect     );     /* outgoing connections */
counter (l4cnt_openrcv     );     /* incoming connections */
counter (l4cnt_sendmsg     );     /* messages sent */
counter (l4cnt_rcvmsg      );     /* messages received */
counter (l4cnt_sendpkt     );     /* packets sent */
counter (l4cnt_rcvpkt      );     /* packets received */
counter (l4cnt_discardfmt  );     /* bad format packets discarded */
counter (l4cnt_discardcon  );     /* bad connect packet discarded */
counter (l4cnt_discardseq  );     /* bad sequence packets discarded */
counter (l4cnt_discardunx  );     /* unexpected packets discarded */
counter (l4cnt_discardrcv  );     /* rcv packet timeout discard */
counter (l4cnt_badwks      );     /* incoming connects on wrong wks */
counter (l4cnt_m_retries   );     /* send message retry attempts */
counter (l4cnt_p_retries   );     /* send packet retry attempts by Level 2 */
counter (l4cnt_aborts      );     /* connections aborts */
counter (l4cnt_abortsends  );     /* connection aborts due to send retries w/o ack */
counter (l4cnt_abortrcvs   );     /* connection aborts due to rcv timeouts */
counter (l4cnt_l2int_ri    );     /* Level 2 receive unsigned interrupts */
counter (l4cnt_l2int_ta    );     /* Level 2 transmit interrupts */
counter (l4cnt_l2int_recon );     /* Level 2 recon interrupts */
counter (l4cnt_xmittimeout );     /* Level 2 transmit timeouts */
counter (l4cnt_xmitnoack   );     /* Level 2 transmit w/o ack (TA w/o TMA) */

/* end of l4counts.h */
```

```
L           JJJ      SSSS
L            J      S   S
L            J      S
L            J       SSS
L            J          S
L   L     J  J      S   S
LLLLL      JJJ      SSSS
```

Wed 30-Apr-1986 12:49:56

Print request number 141

Station: $36

Name: L J Shustek

File Server: BEETHOVEN ($F2)

NFS Pathname:

Filename (s):

```
Print Server: LENNON ($8A)
     Printer: LASER
       Setup: LANDSCAPE
    Priority: Standard
      Copies: 1
       Eject: 0
```

```
    @@@@    @@@@@@@@@  @@@@@@@@@@@@@@@@@@@@@@@@      @@@@@@@
    @@@@    @@@@@@@@  @@@@@@@@@@@@@@@@@@@@@@@@@@     @@@@@@@@
   @@@@@@  @@@       @@@@          @@@    @@@@@@    @@@   @@@@
   @@@@@@  @@@        @@@         @@@@   @@@ @@@    @@@   @@@@
   @@@@@@ @@@@@@@@@   @@@@@@      @@@   @@@   @@@   @@@@@@@@@
  @@@  @@@@@@@@@@@      @@@@@@   @@@@  @@@@@@@@@@  @@@@@@@@@
  @@@  @@@@@               @@@   @@@  @@@@@@@@@@@  @@@  @@@
 @@@@  @@@@@               @@@@  @@@ @@@      @@@@ @@@   @@@
 @@@    @@@@@@@@@@@@@@@@@@@@@@  @@@ @@@        @@@@@@    @@@
@@@@    @@@@@@@@@@@@@@@@@@@@@   @@@@@@         @@@@@@    @@@
```

```
/* this is l4.c */


/*
                ****************************************
                *                                      *
                *    Fileserver prototype - TRANSPORT  *
                *    ********************************  *
                *                                      *
                *           Module "l4.c"              *
                *                                      *
                ****************************************

            ----------- NESTAR CONFIDENTIAL -----------------


This file contains the implementation of a simple but efficient
subset of the XNS Tranport Protocol designed for dedicated servers.
It's characteristics are:


        * Sequenced Packet Protocol only
        * Multiple simultaneous connections
        * Supports Arcnet and Token Ring for datalink level (L2)
        * Half duplex transmission only on each connection
        * Does not support system-packet connection
        * Does not support gateway routing
        * Does not support out-of-sequence packet processing
          (Implies a server that does multiple receives)
        * Supports selective socket listening, but not socket
          demultiplexing.  Any incoming connection can be returned
          to any request.

Some of these restrictions might be removed without excessive effort,
but they do not affect operation of the server.

Note that the fact that packets are processed in the order in which
they are received both makes the implementation simpler and matches
the inability of the TI Token Ring chipset to provide out-of-sequence
packet processing.

*/

/*-------------------------------------------------------------------------------

Change log


02/xx/85   L. Shustek   Initial design document.
02/xx/85   J. Whitnell  Iterations and refinement of design document.
10/29/85   L. Shustek   Started coding, for toy fileserver.
11/30/85   L. Shustek   Resume coding.  Start debugging.
 1/23/86   L. Shustek   Resume debugging.
 1/24/86   L. Shustek   Fix assignment of saved buffer pointer for l4_gotbuf.
 1/28/86   L. Shustek   Log incoming connects as unclaimed packets until the openrcv.
 2/ 7/86   L. Shustek   Major change to incoming packet processing and rb queuing
                        to deal with delayed duplicates of initial connections.
                        Level 4 now keeps track of all rbs with open connections.
 2/13/86   L. Shustek   Delay connect-message ack until the incoming packet has
                        been consumed, to avoid deadly embrace clogging buffers.
 2/20/86   L. Shustek   Add support for short-packet-only connections,
                        primarily to be able to talk to Apple ]['s.
 2/22/86   L. Shustek   For Intel processors, use an assembly routine instead
                        of C to reverse XNS header words.
 2/25/86   L. Shustek   Allow zero-length messages.  Required extra boolean in
```

```
                          parmlist of l4_fillpkt();
 3/19/86   L. Shustek   Up and running on the 68000 now; almost no changes!
                          Fix padding of packets to even number of bytes.
                          Allow connect with zero-length message.
 4/10/86   L. Shustek   Comment out the pktsize trace entry.
 4/15/86   L. Shustek   Crank down timeouts to reasonable production values.

-------------------------------------------------------------------------------*/

#include "exec.h"              /* Realtime exec symbols */
#include "mon.h"               /* Debugging monitor symbols */
#include "l4.h"                /* Public L4 symbols */
#include "l4private.h"         /* Private L4 symbols and variables */

#define L4GLOBALS
#include "l4counts.h"          /* Allocate L4 counters */
```

/***

On calling Level 4 Transport Routines

****

The public Level 4 routines are as follows:

```
l4_init ()          Initialize level 4
l4_terminate ()     Terminate level 4
l4_listen (wks)     Register a well-known socket
l4_unlisten (wks)   Unregister a well-known socket
l4_connect (rb)     Establish an outgoing connection
l4_openrcv (rb)     Wait for an incoming connection
l4_sendmsg (rb)     Send a message
l4_rcvmsg (rb)      Receive a message
l4_disconn (rb)     Disconnect the connection
l4_abort (rb)       Abort the connection
```

Most of the L4 routines are passed only a single argument, which is the address of the caller-allocated L4 Request Block (L4RB). All input and output parameters are exchanged inside the L4RB. See the individual routine descriptions for the details of what is expected and returned.

The "status" field of the L4RB is always valid, and indicates the state of the connection and/or the previous command. The possible values are:

```
l4st_uncon      There is no connection established.
                Either there never was one, or a previous connection
                was terminated because of a call to l4_disconn,
                l4_abort, or an unrecoverable error.

l4st_busy       A previous command is still in progress.  No calls to
                Level4 routines for this rb are valid except for
                l4_abort.

l4st_done       The previous command has completed successfully.
                The connection is still established.
```

There is a class of routines which accomplish their task immediately and status is valid upon return. An example is "l4_listen" which established a well-known socket to listen on.

The other class of routines may not complete immediately, and depend on subsequent hardware interrupts. Those routines return "in progress" status in the L4RB. When the interrupt which completes the command occurs, the status is changed to one of the other status values and the caller's Asynchronous Notification Routine (ANR), whose address is in the L4RB, is called with the address of the L4RB as its only parameter.

Note that the ANR is called from the interrupt environment, so it should execute quickly and be extremely circumspect as to the use of global data structures. You may need to disable interrupts during non-ANR code which access such data structures to keep ANR routines from executing. ANR routines may not call any L4 routines.

It is possible that a routine which usually calls an ANR may be able to complete without waiting for an interrupt. In that case the ANR is called directly from the routine which initiates the command, after which the command routine will return. Beware of subtle timing of the interrupt which calls the ANR when writing the code which checks for completion of a command. Aren't asynchronous systems fun?

The detailed descripion of the input/output parameters and behavior of each routine is located at the entry point to the routine.

*/

/***

Notes on the implementation

****

Header notes
----

1. The following is the full Internet Header and a discussion of the fields. Some of this information is in the silver book ("Internet Transport Protocols", a Xerox System Integration Standard) and is rehashed here. Other parts represent Nestar-specific fields and uses.

a. The header format, not included Arcnet or Token-ring datalink headers:

```
        0 1                           15

        +---------------+-----------------+
  -4    | sysid = $FE       garbage cnt   |
        +---------------+-----------------+
  -2    | packet number    fragment number|
        +---------------+-----------------+

        +---------------------------------+ <-----------------+
   0    |            Checksum             |                   |
        +---------------------------------+                   |
   2    |             Length              |                   |
        +-------------------+-------------+                   |
   4    | Transport Control | Packet Type |                   |
        +-------------------+-------------+
   6    |           Destination           |
        +----         Network        -----+
   8    |                                 |
        +---------------------------------+
  10    |           Destination           |
        +----          Host          -----+
  12    |                                 |
        +----                        -----+
  14    |                                 |            Network Layer
        +---------------------------------+              (Level 3)
  16    |       Destination Socket        |
        +---------------------------------+
  18    |             Source              |
        +----         Network        -----+
  20    |                                 |
        +---------------------------------+
  22    |             Source              |
        +----          Host          -----+
  24    |                                 |                   |
        +----                        -----+                   |
  26    |                                 |                   |
        +---------------------------------+                   |
  28    |          Source Socket          |                   |
        +---------------------------------+ <-----------------+
  30    |         Level 4 header          |
        |          (see below)            |
```

i.   Sysid - The Datapoint-administered system protocol identifier. The value assigned to Nestar protocols is FE.
ii. Garbage cnt - The count of extra bytes that were added at the end of the packet for datalink-dependent padding.
iii. Packet number - A sequential packet number used for packets fragmented by gateways.
iv. Fragment number - The fragment number within a packet, used for packets fragmented by gateways.

a. Checksum - Checksum of level 3 packet. FFFF means not checksumed.
b. Length - Length of Internet packet including checksum.
c. Transport Control - For use by internetwork routers. Always 0 for clients.
d. Packet Type - Type of Level 4 packet being sent. Types include 5 for Sequence Packet.
e. Network Addresses - A network address consists of three parts. The Host Number is a unique in all space and time 48 bit address for a station . The Network Number designates which individual network of the Internetwork the station is attached too. Socket number is a bidrectional structure capable of sending and recieving packets at the same address. Certain sockets are "well-known", which means they are known by other stations.

   The Source Network Address is the address from which the packet originated. The Destination Network Address is the address to which the packet must be delivered.

```
        0 1                            15
        +---------------+-----------------+  <-----------------+
   30   | Conn Control  | Datastream Type |                    |
        +---------------+-----------------+                    |
   32   |      Source Connection Id       |                    |
        +---------------------------------+                    |
   34   |    Destination Connection Id    |                    |
        +---------------------------------+
   36   |         Sequence Number         |        Sequenced Packet
        +---------------------------------+       Protocol (Level 4)
   38   |       Acknowledge Number        |
        +---------------------------------+
   40   |        Allocation Number        |                    |
        +---------------------------------+  <-----------------+
   42   |                                 |
        |          Level 5 data           |
```

f. The Conn Control consists of four bits (0 - 3) that control the action of the protocol and four bits (4 - 7) that are unassigned and should be 0. The System Packet bit (bit 0) indicates that this packet contains no data and does not consume a sequence number. The Send Acknowledgment bit (bit 1) indicates the receiver should send back an acknowledgment. The Attention bit (bit 2) indicates that the sender desires immediate notification that this packet arrived. Only 1 byte of data can be included in a packet with the Attention bit set. The End Of Message bit (bit 3) indicates the boundry of a message.
g. The Datastream Type is a level 5 type passed in the level 4 header
h. The Connection Ids are unique Identifiers allocated by each machine at the beginning of a connection to uniquly identify the connection.
i. The Sequence Number counts packets sent during a connection. Each direction has its own sequence number.
j. The Ackowledge Number specifies the sequence number of the first packet which has not yet traveled in the reverse direction.
k. The Allocation Number specifies the sequence number up to and including which packets will be accepted from the other end. Said another way, one plus the difference between the Allocation Number and the Acknowledge Number indicates the number of packets that may be outstanding in the reverse direction.

2. Migration. Migration is process of moving from the well-known socket used to establish to connection to a temporary ephemeral socket number. There are two things that migrate from a well-known socket to a ephemeral socket: The socket number of the 14 doing the open_recv in the request block and and the same socket number in the packet(s) coming in for that connection. The open_receiver cannot do it until the connector is informed that the change has taken place, in order to handle retries of the connect request. Furthermore, the open_receiver must handle packets on the well-known socket until the sender sends a packet on the ephmeral socket the open_receiver has moved the connection to. So we see:

```
C--- e_wks -->OR
 --- e_wks -->
 <--- e_e ----
 ---- e_e --->
```

So the connector must migrate the connection at the point an ack from the receiver comes back with the ephemeral socket. And the open_receiver cannot migrate until the connector sends a packet on the ephemeral socket. So the proper place to do the migration for the connector is on the reception of a packet with a non-well-known socket. The open_receiver needs to do it when a packet is received back from the connector on a well-known socket. Note we can look at the two places of migration as the open_receiver's idea of what to send to the other end (for the rb) and what to receive from the other end (for the packet).

Our original IBMPC level 4 (CWP's) migrates the rb end when a packet comes in for an rb in the state conn_accept_wait. It migrates the other end as soon as open_recv receives a packet. Retries are apperantly not allowed on connection (i.e. they either get assigned to an open_recv or get tossed).

An interesting question here is what about multiple open_receives on the same socket. Once a packet matches up with an rb, how are retries of that packet to match up with that rb and not some other rb doing an open_recv? Obviously, matching of packet headers depends on the state the rb is in. We may not want to allow multiple open_recvs on a single socket.

3. The distinction between the datalink (arcnet) address from which packets are received and the "source host" XNS address must be maintained so that connection to stations through a gateway will work. It's really simple: all the XNS processing is based on XNS addresses, but the arcnet address used at the last minute is from the datalink address field of the RB.

4. Our level 4's always set the ack request bit of a connection. This will cause the open_receiver to generate a piggy-backed ack if a send_msg is done following the open_recveive or a system packet ack if a recv_msg or ack_now (an optimization for cases where neither a send_msg or recv_mesg is done soon) is done. Ack request is also set when EOM is set.

Internal Structure Notes

1. Request Blocks

The repository for information about a connection is called a Request Block (RB). In addition to the externally-visible "status" field which gives completion information to the caller, there is an internal "state" variable which indicates the internal phase of the process. Note there is not necessarily a value for "state" for each internal phase, but only phases during which the processing for the rb might be suspended awaiting an interrupt event.

2. The status of RBs

Allocation and deallocation of RBs is done by the caller. While an RB has been passed to Level4 to execute a command the status variable value is "l4st_busy". Whenever Level4 is not executing, each RB in its care (in the "l4st_busy" state) is on one of following queues:

```
    waitbuf_list    RBs waiting for a free transmit buffer
    waitpkt_list    RBs connected and waiting for a packet to come in
    waitcon_list    RBs not yet connected and waiting for a connect
                    packet to a well-known socket.
```

These lists are doubly linked with a head and tail pointer. The double linking is so that deletions from the middle of the waitpkt list when the packet arrives is quick. Insertions are made at the tail so that the RBs are processed in FIFO order coming off the waitbuf list.

In addition, any RB which has a connection established is also registered in the connection table. If the RB is waiting for an incoming packet, the rb.state indicates what sort of packet it is waiting for, and the the RB is on the waitpkt_list. The connection table is used to find the RB when an appropriate packet arrives, but the waitpkt_list is used by the timer interrupt to see if a timeout should be triggered because the RB has been waiting too long.

3. Internal RB state

There is a single rb->state variable, plus some associated flags. See the declarations in l4private.h for more information about the states.

The sequence of state transitions is roughly as shown in the following diagram. Note that an RB may remain in some of the states through several interrupts in order to get it's job done (such as st_sending when there are multiple packets to go out) or may skip a state entirely if it is unnecessary (such as st_sendackd if there is no ack owed.)

```
st_idle

   |
   |
   |  l4_connect
   |----------------->  st_connectg  ----------->  st_ackwait ----->  st_idle
   |                         A                          |
   |                         |      more to send        |
   |                     <---------------------------------
   |                     |   |
   |                     |   |
   |                     |   |        (retry)
   |                     |   |<-------------------
   |                     |   |                    |
   |                     |   |                    |
   |  l4_sendmsg         V   V                    |
   |----------------->  st_sending  ---------->  st_ackwait  ------>  st_idle
   |
   |
   |  l4_openrcv
   |----------------->  st_connwait  --------->   st_idle
   |
   |
```

```
 |                                -------------------------
 |                                |                       |
 |                                |       ackreq          |
 |   l4_rcvmsg                    V       & !eom          |
 |----------------->    st_pktwait   ------------>  st_sendackr
 |                               |  |
 |                               |  | connect ack
 |                               |  --------------->  st_sendackc   ----->  st_idle
 |                               |
 |                               |   eom
 |                               ------------------------------------->  st_idle
 |
 |   l4_disconn
 |----------------->    st_sendackd   ----------------------------->  st_idle
```

4. The absence of polling

This implementation of Level4 is interrupt-driven and there is no polling for events. In addition, the separate lists of RBs awaiting events are designed to minimize searching for RBs so that it is efficient even with a large number of active connections in progress. A hash table is used to find the RB to which an incoming packet should be assigned, so that too is fast.

5. Management of timeouts

In addition to timeouts for RBs on one of the waiting lists, incoming and outgoing packets may have to be timed out with the aid of interrupts from a hardware timer. There are two general schemes that could be used:

a. Keep a time-ordered queue of timeout events. The soonest event on the queue is at the head of the list, and the hardware timer is programmed to interrupt at that time.

b. Each event contains a countdown timer word. The hardware timer is programmed to interrupt periodically, at which time the countdown word for each event is decremented. If a countdown word reaches zero, a timeout has occurred.

The characteristic of timeouts as used by Level4 is that the initiation of a timeout interval is a very frequent event (every time a packet is expected, for example), but timeout intervals are long and the occurence of a timeout is rare. Although the second scheme for handling timeouts is a violation of the non-polling dictum, it is much more efficient because it avoids insertion and deletion in an ordered list when timeout intervals are established and cancelled.

To reduce the overhead from the periodic interrupt, the period is chosen as about 1/2 or 1/3 of the minimum timeout value. It can't be the minimum timeout value because the interrupt is asynchronous, and specifying a timeout of 1 means that it might go to zero in an arbritrarily small time. An interrupt period of about 200 msec seems right, and results in a tolerable overhead (1 msec, say, out of 200 = 0.5%).

6. The management of interrupts

The Level 4 interrupt routines which are described in the next section must be delayed if Level 4 is currently active because they are part of Level 4 and the same data structures (RBs, waiting lists, etc.) are manipulated. The global Level 4 flag "l4_busy" is used as a semaphore to delay the interrupt routine, and each synchronous routine checks for a pending asynchronous interrupt request when turning off the flag.

The result is that hardware interrupts need not be disabled at any time during the execution of Level 4, except briefly as the busy flag is turned off.

The following sketch represents how interrupts are delayed:

```
    interrupt()

        if busy
                assert: not int_pending
                int_pending = TRUE
                return
        busy = TRUE
        .... proceess interrupt ....
        busy = FALSE
        return


    request routine()

        assert: not busy
        busy = TRUE
        .... process request ....
        busy = FALSE
        while int_pending                   *
                int_pending = FALSE         *
                interrupt()
        return
```

Note that hardware interrupts which occur during the execution of the instructions marked with a '*' may be processed before an earlier pending interrupt. This out-of-sequence processing of interrupts does not cause any problems in L4, but should be kept in mind.

*/

```
/*-----------------------------------------------------------------------------

            Level 2 packet routine definitions

-----------------------------------------------------------------------------*/

/*  The following routines are entries into Level 2, called by Level 4 */


extern faraddr l2_getbuf();

    /*  Get an empty transmit buffer and return the address ofit.
        If there aren't any free buffers, return NIL and call the
        routine  l4_gotbuf  as an interrupt routine when there is one. */


extern l2_sendbuf ( /* faraddr */ );

    /*  Send, or queue for sending, the transmit buffer whose
        address is supplied. */


extern l2_rcvrelease ( /* faraddr */ );

    /* Release the received packet whose buffer address is supplied.
       It was previously provided by a call to l4_rcvintr.
       This could cause l4_rcvintr to be called if another packet is ready. */


extern boolean l2_init ( /* &our_addr */ );

    /*  Initialize Level 2, and return our address.
        Return FALSE if initialization failed. */


extern l2_terminate ();

    /*  Terminate Level 2. */



/*  The following routines are entries into Level 4, called by Level 2
    interrupt routines.  See the commentary at each routine for more
    details.

l4_gotbuf (faraddr);       "Have empty buffer" interrupt routine.

l4_rcvintr (faraddr);      "Received packet" interrupt routine.

l4_timerint ();             "Timer tick" interrupt routine.

*/
```

```
/*---------------------------------------------------------------------------


            Other external routines used


-----------------------------------------------------------------------------*/

/*
    Move "count" bytes from a1 to a2.

    For Microsoft C on the 8088, both addresses are far (segmented), so
    this is a different routine than move().
    For Microtec C on the 68000, movel()is functionally the same as move()
    but separated so that the histogram identifies packet buffer moves
    separately from other moves.
*/

extern movel ( /* faraddr a1, faraddr a2, int count */ );


/*


14_trace ( int event, info1, info2 );

    This is a macro which records the occurence of "event" with
    optional information info1 and info2.  If this event is
    associated with an rb, info1 is the address of the rb.



assert (boolean e, *char string);

    This is a macro which asserts that "e" had better be true,
    or else we panic stop and display the "string".

*/
```

```
/*----------------------------------------------------------------------------

boolean l4_init ()

Initialize Level 4 Transport
Return TRUE if it succeeds.

----------------------------------------------------------------------------*/

boolean l4_init ()

{
short int i;

   for (i=0; i < MAX_SOCKETS;  ++i)      /* initialize all well-known sockets */
       sockets[i] = -1;

   for (i=0; i < HASH_TABLE_SIZE; ++i)  /* initialize the connection hash table */
        l4_con_table[i] = NIL;


    /* The following intialization is necessary only if level 4 is
       restarted, since these globals are compiler-initialized to
       the correct value.

       Event counters are expected to be zeroed by someone else.
    */

    l4_busy = FALSE;
    waitbuf_list.head = waitbuf_list.tail = NIL;
    waitpkt_list.head = waitpkt_list.tail = NIL;
    waitcon_list.head = waitcon_list.tail = NIL;
    rcvbuf_xptr = NIL;   rcvbuf_rb = NIL;
    rcvbuf_timer = 0;
    intpending_rcv = intpending_xmit = intpending_timer = FALSE;


    /*  Initialize level 2 and return */

    return l2_init ( our_addr );
    }




/*----------------------------------------------------------------------------

l4_terminate ()

Terminate Level 4 Transport

----------------------------------------------------------------------------*/

l4_terminate ()

{
    l2_terminate ( );       /* just terminate level 2 */
    }
```

```
/*-----------------------------------------------------------------------------

14_listen (wks)

short int wks;

Allow incoming connections on the specified well-known socket.
Return immediately without executing an ANR.

If no more listen sockets are available, generate an internal error.

-----------------------------------------------------------------------------*/

14_listen (wks)

short int wks;

{ short int i;

   for (i=0; i < MAX_SOCKETS;  ++i)   /* see if is already being listened on */
       assert (sockets[i] != wks,"14_listen 1");  /* error if so */

   for (i=0; i < MAX_SOCKETS;  ++i) {  /* find a free socket slot */
       if (sockets[i] == -1) {
           sockets[i] = wks;            /* and use it */
           return;}
       }
   assert (FALSE,"14_listen 2");       /* no free listen socket slots */
   }


/*-----------------------------------------------------------------------------

14_unlisten (wks)

short int wks;

Disallow incoming connections on the specified well-known socket.
Return immediately without executing the ANR.

-----------------------------------------------------------------------------*/

14_unlisten (wks)

short int wks;

{ short int i;

   for (i=0; i < MAX_SOCKETS;  ++i) {
       if (sockets[i] == wks) {  /* if the socket matches, */
           sockets[i] = -1;      /* mark it free */
           }
       }
   }
```

```
/*------------------------------------------------------------------------------

14_connect (&14rb)


Start a connection to a remote station by sending an initial message.


Inputs are the following 14rb fields:

   ph.dsthost = the 6-byte destination station address
   ph.dstskt  = the 2-byte destination well-known socket
   sndptr     = a pointer to the connect message to send
   sndlength  = the length of the connect message
   sndtype    = the 1-byte type of the connect message
   status     = 14st_uncon to indicate this is an unused rb
   arcnet     = TRUE if this is an arcnet station, FALSE for token ring
   anr        = the function to call when the connect is complete

All other rb fields must be zeroed!


Output is the status field of the 14rb, as follows:

   status == 14st_uncon    The connection failed.
   status == 14st_done     The connection succeeded and the link
                           is established.

Until the command is complete the status field will be 14st_busy.
The ANR routine whose address is in the 14rb will be called when
the status is changed to one of the above.

------------------------------------------------------------------------------*/

14_connect (rb)

register struct 14rb *rb;

{
short int hash_index;       /* index into connection hash table */

    14_trace(tr_14connect,rb,
        eph_socket);             /* log the connect call */
    ++14cnt_connect;             /* count it as an outgoing connection */
    ++14cnt_sendmsg;             /* count it as a message */

    assert (rb->id == RBid, "14_connect 0");
    assert (rb->status == 14st_uncon,"14_connect 1");
    assert (!14_busy,"14_connect 2");
    assert (rb->sndlength>=0, "14_connect 3");

    14_busy = TRUE;              /* 14 is busy */
    rb->status = 14st_busy;      /* command is in progress on this rb */

    rb->state = st_idle;         /* force idle state */


    /* Link this rb into the connection hash table */

    hash_index = hash_addr(rb->ph.dsthost);
    rb->conlink = 14_con_table [hash_index];    /* add it at the head */
    14_con_table [hash_index] = rb;
```

```
/* start sending */

14_initph(rb);                          /* initialize the packet header */
rb->send_ack = FALSE;                   /* no special ack with eom */
rb->ph.dtype = rb->sndtype;             /* copy datastream type */
rb->ph.dstid = 0;                       /* destination id is unknown */
rb->ph.maxpkt = MAXPKT_LONG;            /* offer long packet support */
rb->first_seq = 0;                      /* save our starting sequence number */
rb->snd_count = 0;                      /* no message retries yet */
rb->bufcursor = rb->sndptr;             /* initialize cursor to start of buffer */
rb->ph.allno = rb->ph.seqno + 100;      /* arbitrary large allocation number */
rb->bytes_left = rb->sndlength;         /* amount to send */
rb->state = st_connectg;                /* put us in the connecting state */
14_dosend(rb, 12_getbuf());             /* process send until blocked */

14_exit();  /* turn off 14_busy flag and process pending interrupts */
}
```

```
/*----------------------------------------------------------------------------

14_openrcv (&14rb)

Wait for an incoming connection for any well-known socket
we are listening to.


Inputs are the following 14rb fields:

   anr        = routine to call when a message is incoming
   status     = 14st_uncon

All other rb fields must be zero.


When a connection has been received and an ack sent, the ANR
routine will be called.  When that occurs, 14_rcvmsg
should be called to supply a buffer for the message.

Outputs are the following rb fields:

    status     = 14st_done
    ph.dsthost = the 6-byte host address of the other station

-----------------------------------------------------------------------------*/

14_openrcv (rb)

register struct 14rb *rb;

{
    14_trace(tr_14openrcv,rb,0);   /* log the call to openrcv */

    assert (rb->id == RBid, "14_openrcv 0");
    assert (rb->status == 14st_uncon ,"14_openrcv 1");
    assert (!14_busy,"14_openrcv 2");
    assert (rb->state == st_idle, "14_openrcv 3");

    14_busy = TRUE;                      /* 14 is busy */
    rb->status = 14st_busy;              /* command is in progress on this rb */

    if (rcvbuf_xptr                      /* if there is a valid packet waiting */
    &&  rcvbuf_rb == NIL                 /* and not assigned to anyone */
    &&  rcvbuf_xptr->dstskt < MAX_WKS){  /* and it is a new connection */
        rcvbuf_rb = rb;                  /* grab the packet */
        14_newconn(rb);                  /* then process it now */
        }

    else { /*  There isn't an incoming connection available.
               Put us on the list for incoming connections and return. */

        rb->state = st_connwait;  /* awaiting a connection */
        14_addlist(rb, &waitcon_list);
        }

    14_exit();  /* turn off 14_busy flag and process pending interrupts */
    }
```

```
/*----------------------------------------------------------------------------

14_sendmsg (&14rb)


Send a message on an existing open connection.


Inputs are the following 14rb fields:

   sndptr     = a pointer to the message to send
   sndlength  = the length of the message
   sndtype    = the 1-byte type of the message
   status     = 14st_done to indicate this is idle open connection
   anr        = function to call when the send is complete

All other rb fields must be unchanged.


Output is the status field of the 14rb, as follows:

   status == 14st_uncon    The send failed and the connection is closed.
   status == 14st_done     The send succeeded and the connection is still
                           open.

Until the command is complete the status field will be 14st_busy.
The ANR routine whose address is in the 14rb will be called when
the status is changed to one of the above.

----------------------------------------------------------------------------*/

14_sendmsg (rb)

register struct 14rb *rb;

{
    14_trace(tr_14sendmsg,rb,
        rb->ph.seqno);              /* log the call to sendmsg */
    ++14cnt_sendmsg;                /* count it */

    assert (rb->id == RBid, "14_sendmsg 0");
    assert (rb->status == 14st_done,"14_sendmsg 1");
    assert (!14_busy,"14_sendmsg 2");
    assert (rb->state == st_idle, "14_sendmsg 3");
    assert (rb->sndlength>=0, "14_sendmsg 4");

    14_busy = TRUE;                 /* 14 is busy */
    rb->status = 14st_busy;         /* command is in progress on this rb */

    if (rcvbuf_rb == rb) {          /* if we own a packet, discard it */
        ++14cnt_discardunx;         /* count discard of unexpected packet */
        14_trace(tr_14discardunx,rb,
              rcvbuf_xptr->seqno);  /* log it */
        12_rcvrelease (rcvbuf_aptr);/* discard the packet */
        rcvbuf_xptr = NIL;  rcvbuf_rb = NIL;
        }

    /* start sending */

    rb->ph.dtype = rb->sndtype;         /* datastream type */
    rb->snd_count = 0;                  /* no message retries yet */
    rb->first_seq = rb->ph.seqno;       /* save starting sequence number */
    rb->bufcursor = rb->sndptr;         /* initialize cursor to start of buffer */
    rb->ph.allno = rb->ph.seqno + 100;  /* arbitrary large allocation number */
```

```
rb->bytes_left = rb->sndlength;     /* amount to send */
rb->state = st_sending;             /* put us in the sending state */
l4_dosend(rb, l2_getbuf());         /* process send until blocked */

l4_exit();  /* turn off l4_busy flag and process pending interrupts */
}
```

```
/*---------------------------------------------------------------------------

14_rcvmsg (&14rb)

Receive a message on an existing open connection.

Inputs are the following 14rb fields:

   rcvptr      = a pointer to the message buffer
   rcvlimit    = the size of the message buffer
   status      = 14st_done to indicate this is idle open connection
   anr         = function to call when the send is complete

All other rb fields must be unchanged.

Output is the status field of the 14rb, as follows:

   status == 14st_uncon    The receive failed and the connection is closed.
   status == 14st_done     The receive succeeded and the connection is still
                           open.

Until the command is complete the status field will be 14st_busy.
The ANR routine whose address is in the 14rb will be called when
the status is changed to one of the above.

When status == 14st_done, the following additional fields will have been set:

    rcvlength   ==  the actual length of the received message
    rcvtype     ==  the type of the received message
---------------------------------------------------------------------------*/

14_rcvmsg (rb)

register struct 14rb *rb;

{
    14_trace(tr_14rcvmsg,rb,
        rb->ph.ackno);              /* log the call to rcvmsg */
    ++14cnt_rcvmsg;                 /* count it */

    assert (rb->id == RBid, "14_rcvmsg 0");
    assert (rb->status == 14st_done,"14_rcvmsg 1");
    assert (!14_busy,"14_rcvmsg 2");
    assert (rb->state == st_idle, "14_rcvmsg 3");
    assert (rb->rcvptr, "14_rcvmsg 4");

    14_busy = TRUE;                 /* 14 is busy */
    rb->status = 14st_busy;         /* command is in progress on this rb */

    rb->bufcursor = rb->rcvptr;     /* start the buffer pointer */
    rb->rcvlength = 0;              /* start the cumulative length */

    if (rcvbuf_rb == rb) {        /* There is already a packet of data assigned to us. */
        14_processpkt (rb);       /* use it */
        }

    else {                        /* there is no packet yet */
```

```
        rb->state = st_pktwait;
        rb->timer = TO_AWAIT_MSG;
        14_addlist (rb, &waitpkt_list);  /* wait for the initial packet */
        }

/* Note that if there was an initial packet but we disarded it because it
   smelled funny, we will be on the waitpkt_list with the TO_AWAIT_PKT
   timeout instead of the TO_AWAIT_MSG timeout.  Not perfect, but so what.
*/

    14_exit();  /* turn off 14_busy flag and process pending interrupts */
    }
```

```
/*------------------------------------------------------------------------

    l4_abort (&l4rb)

Abort the current connection.

This is for external callers.
------------------------------------------------------------------------*/

l4_abort (rb)

register struct l4rb *rb;

{
    assert (rb->id == RBid, "l4_abort 0");

    l4_busy = TRUE;
    l4_doabort(rb);
    l4_exit();  /* turn off l4_busy flag and process pending interrupts */
    }


/*------------------------------------------------------------------------

    l4_doabort (&l4rb)

Abort the current connection.

This is for internal callers.
------------------------------------------------------------------------*/

l4_doabort (rb)

register struct l4rb *rb;

{

    l4_trace(tr_l4abort, rb, 0);        /* make a log entry */
    ++l4cnt_aborts;                     /* count it */

    l4_removecon (rb);                  /* remove it from the connection table */
    l4_purgelist (rb, &waitbuf_list);   /* purge from any lists it is on */
    l4_purgelist (rb, &waitpkt_list);   /* purge from any lists it is on */
    l4_purgelist (rb, &waitcon_list);   /* purge from any lists it is on */
    assert (!rb->on_a_list, "l4_abort 1");

    rb->status = l4st_uncon;            /* unconnected */
    rb->state = st_idle;                /* and idle */

    if (rcvbuf_rb == rb) {              /* if we own the current incoming packet, */
        l2_rcvrelease (rcvbuf_aptr);    /* discard it */
         rcvbuf_xptr = NIL; rcvbuf_rb = NIL;
        }
    }
```

```
/*--------------------------------------------------------------------------

    l4_disconn (&l4rb)

Disconnect the current connection.

---------------------------------------------------------------------------*/

l4_disconn (rb)

register struct l4rb *rb;

{
    l4_trace(tr_l4disconn,rb,
        rb->ph.srcskt);                /* log the l4_disconn call */

    assert (rb->id == RBid, "l4_disconn 0");
    assert (rb->status == l4st_done,"l4_disconn 1");
    assert (!l4_busy,"l4_disconn 2");
    assert (rb->state == st_idle, "l4_disconn 3");

    assert (!rb->on_a_list, "l4_disconn 4");  /* we should be on no lists */

    if (rcvbuf_rb == rb) {           /* if we own a packet, discard it */
        ++l4cnt_discardunx;          /* count discard of unexpected packet */
        l4_trace(tr_l4discardunx,rb,
            rcvbuf_xptr->seqno);  /* log it */
        l2_rcvrelease (rcvbuf_aptr);/* discard the packet */
        rcvbuf_xptr = NIL;  rcvbuf_rb = NIL;
        }

    l4_busy = TRUE;                    /* l4 is busy */
    rb->status = l4st_busy;            /* command is in progress on this rb */

    if (rb->we_owe_ack) {  /* we owe him an ack first */

        rb->state = st_sendackd;       /* "we are sending an ack for disconnect" */
        assert (rb->bytes_left == 0, "l4_disconn 6");
        l4_dosend(rb, l2_getbuf());    /* try to send it */
        }

    else {  /* we can disconnect right now */

        rb->status = l4st_uncon;       /* we are unconnected */
        rb->state = st_idle;           /* and idle */
        l4_removecon (rb);             /* remove it from the connection table */
        l4_trace(tr_l4disconned,rb,
            rb->ph.srcskt);            /* log the "disconnnected" anr call */
        (*rb->anr)(rb);                /* call the "disconnected" ANR */
        }


    l4_exit();  /* turn off l4_busy flag and process pending interrupts */
    }
```

```
/*---------------------------------------------------------------------------

    l4_exit ()

Exit from a public Level 4 routine.

Check for any pending interrupts that were postponed.

---------------------------------------------------------------------------*/

l4_exit ()

{

    l4_busy = FALSE;

    /* We must loop until all the pending flags are off because
       processing a delayed interrupt sets l4_busy and a interrupt
       which occurs at that time would itself become pending and
       not be caught if we had already looked at that flag and
       found it false.  (Thanks, Jerry!)
    */

    while (intpending_rcv || intpending_xmit || intpending_timer) {

        /* Note that because we do these checks while interrupts are enabled
           and the l4_busy flag is off, there is a small chance that an
           interrupt can occur right now and be processed out of order.
           But there is no harm in that, so we don't spend the time to disable.
        */

        if (intpending_rcv) {           /* pending receive interrupt */
            intpending_rcv = FALSE;
            l4_rcvintr (intpending_bufp);
            }

        if (intpending_xmit) {          /* pending transmit interrupt */
            intpending_xmit = FALSE;
            l4_gotbuf (intpending_outbuf);
            }

        if (intpending_timer) {         /* pending timer interrupt */
            intpending_timer = FALSE;
            l4_timerint();
            }

        } /* while */
    }
```

```
/*-------------------------------------------------------------------------------

l4_newconn (&l4rb)


The current packet is an incoming connect that can be assigned to
this RB, whose last call was to l4_openrcv.

Record the connection data, assign an ephemeral socket and conid,
and note a special ack to be sent if it is a one-packet message.
Also record whether the "long packets ok" bit is set in the header.

Remember that although the ackreq bit is usually set with eom, we don't
honor it so that the ack will piggyback on the next message.  The "special
connect ack" is an ack sent because of an ackreq with eom if it is the
first message of a connection and it is a single-packet message. If it is
a multi-packet connect, the first packet will have the ackreq bit but not
the eom bit set, and thus generate the ack from l4_processpkt, and so we
cancel the request for the special ack.

Note too that we do not send the special ack until the l4_rcvmsg call has
released this packet.  This is important to avoid a deadly embrace with
two stations who are connecting to each other almost simultaneously. What
happens is that both stations have their incoming buffers clogged with
initial connect packets but are trying to send each other the ack which
will allow them to be processed and unclogged.

-------------------------------------------------------------------------------*/

l4_newconn (rb)

register struct l4rb *rb;

{
short int hash_index;

    ++l4cnt_openrcv;                    /* count an incoming connection */

    hash_index = hash_addr(rcvbuf_xptr->srchost); /* add this rb to the end of the */
    rb->conlink = l4_con_table [hash_index];      /* connection table */
    l4_con_table [hash_index] = rb;

    l4_initph(rb);  /* initialize our packet header for sending */
    rb->ph.dsthost[0] = rcvbuf_xptr->srchost[0];  /* copy his address */
    rb->ph.dsthost[1] = rcvbuf_xptr->srchost[1];
    rb->ph.dsthost[2] = rcvbuf_xptr->srchost[2];
    rb->ph.dstid =      rcvbuf_xptr->srcid;  /* incoming src id is our dst id */
    rb->ph.dstskt =     rcvbuf_xptr->srcskt; /* incoming src skt is our dst skt */
    rb->rcvtype =       rcvbuf_xptr->dtype;  /* preview the datastream type */
    rb->ph.maxpkt =     rcvbuf_xptr->maxpkt; /* copy MAXPKT_LONG bit */

    rb->status = l4st_done;             /* we are done with the openrcv */
    rb->state = st_idle;                /* and are now idle */
    rb->send_ack = TRUE;                /* flag to send ack with eom */
    l4_trace(tr_l4connectrcvd,rb,
                   rb->ph.srcskt);      /* log the "connect rcvd" anr call */
    (*rb->anr)(rb);                     /* call the "openrcv" ANR */
    }
```

```
/*---------------------------------------------------------------------------

14_dosend (&14rb, outbuffer)

The rb is currently in one of the following sending phases:

        st_connectg:  sending the first packet of an intial connect message

        st_sending:   sending packets of a message

        st_sendackr:  sending an ack because we got a packet with the
                      ack-request bit on and the end-of-message bit off.

        st_sendackc:  sending an ack because the only (or last) packet of an
                      incoming connection has arrived.

        st_sendackd:  sending an ack because of an 14_disconn call.

The parameter "outbuffer" is the address of a packet buffer if one is
available, or NIL if there are none at the moment.

This is called for the initial attempt at sending, and by the interrupt
routine which discovers a new transmit buffer if this rb was on the
wait-for-buffer list.

---------------------------------------------------------------------------*/

14_dosend (rb, outbuffer)

register struct 14rb *rb;
faraddr outbuffer;  /* address of the arc-format packet buffer */

{

    /* Wait for a buffer if we weren't given one. */

    if (!outbuffer) {   /* if we didn't get a buffer */
        14_addlist (rb, &waitbuf_list);  /* put us on the list for xmit buffers */
        return;
        }


    /* We now have a buffer to send with. */


switch (rb->state) {


case st_connectg:   /* Send the initial packet of a connect message */

    14_fillpkt (rb, outbuffer,
        /* ackreq */ TRUE, /* syspkt */ FALSE);   /* send with ack request */
    goto ackwait;                                 /* and wait for the ack */


case st_sending:  /* Send, or continue sending, a message */

    assert (rb->bytes_left>=0, "14_dosend 1");

    do {  /* send as much as we can */
        14_fillpkt (rb, outbuffer,
```

```
            /* ackreq */ FALSE, /* syspkt */ FALSE);  /* fill packet and queue it for transmit */
        } while (rb->bytes_left>0  &&  (outbuffer = l2_getbuf()) );

   if (rb->bytes_left == 0) {       /* last packet was queued */

ackwait:                            /* or first packet of connect was sent */
        rb->state = st_ackwait;     /* await ack */
        assert (rcvbuf_rb != rb, "l4_dosend 2");  /* we better not own a packet */
        /* If he sends while we are sending, we could own a packet.  This assertion
        should probably be removed after debugging. */
        rb->timer = TO_AWAIT_ACK;
        l4_addlist (rb, &waitpkt_list);  /* put us on the packet-wait list */
        }

    else {  /* must wait for more buffers */
        l4_addlist (rb, &waitbuf_list);  /* put us on the list for xmit buffers */
        /* remain in the sending state */
        }

    break;


case st_sendackr:   /* sending a requested ack in the middle of a message */

    l4_fillpkt(rb, outbuffer,
        /* ackreq */ FALSE, /* syspkt */ TRUE); /* send the ack */
    rb->state = st_pktwait;            /* go back to wait for more packets */
    rb->timer = TO_AWAIT_PKT;
    l4_addlist (rb, &waitpkt_list);
    break;


case st_sendackd:   /* sending an ack because we are disconnecting */

    l4_fillpkt(rb, outbuffer,
        /* ackreq */ FALSE, /* syspkt */ TRUE); /* send it */
    rb->status = l4st_uncon;           /* we are unconnected */
    rb->state = st_idle;               /* and idle */
    l4_removecon (rb);                 /* remove it from the connection table */
    l4_trace(tr_l4disconned,rb,
        rb->ph.srcskt);                /* log the "disconnnected" anr call */
    (*rb->anr)(rb);                    /* call the "disconnected" ANR */
    break;


case st_sendackc:   /* sending an ack because of an incoming connect */

    l4_fillpkt (rb, outbuffer,
        /* ackreq */ FALSE, /* syspkt */ TRUE); /* send an ack */
    rb->status = l4st_done;             /* now receive is done */
    rb->state = st_idle;                /* and we are idle */
    rb->send_ack = FALSE;               /* no more special ack with eom */
    l4_trace(tr_l4rcvanr,rb,
        rb->ph.srcskt);                 /* log rcvmsg done */
    (*rb->anr)(rb);                     /* call his ANR routine */
    break;

default: assert (FALSE, "l4_dosend 4");   /* wrong state in call to dosend */

} /* switch */

} /* l4_dosend */
```

```
/*-------------------------------------------------------------------------


    l4_initph (rb)


    Initialize the transmit packet header.

    This is called both in preparation for an outgoing connection
    and when an incoming connection arrives.

--------------------------------------------------------------------------*/

l4_initph (rb)

register struct l4rb *rb;

{
    rb->ph.sysid = NESTAR_SYSID;    /* Nestar's protocol id */
    rb->ph.chksum = 0xffff;         /* checksum = -1 means "no checksum" */
    rb->ph.ptype = 5;               /* SPP = sequenced packet protocol */
    rb->ph.srcid = src_conid++;     /* use up the next source connection id */
    rb->ph.srcskt = eph_socket++;   /* use up the next ephemeral socket */
    if (eph_socket > MAX_WKS+1000) eph_socket = MAX_WKS+1;
    rb->ph.seqno = 0;               /* start sending sequence number 0 */
    rb->ph.ackno = 0;               /* expect to receive seq. number 0 */
    rb->ph.srchost[0] = our_addr[0];  /* move our source address */
    rb->ph.srchost[1] = our_addr[1];
    rb->ph.srchost[2] = our_addr[2];
    }
```

```
/*------------------------------------------------------------------------------

14_processpkt (&14rb)

Process an incoming packet whose XNS header is at rcvpkt_xptr.

If the sequence number is not right, discard it.
If it's ok, use the data.
If it's the end of the message, call the ANR.

This is called from 14_rcvmsg without us on any waiting list,
and from the interrupt routine 14_rcvintr with us on the pktwait_list.
We return on the pktwait_list if we must wait for more packets for this
message.

------------------------------------------------------------------------------*/

14_processpkt (rb)

struct 14rb *rb;

{
short int length;
int conctl;     /* connection control byte of the packet */

    conctl = rcvbuf_xptr->conctl;              /* remember the control byte */

    if ( conctl & syspkt                       /* if not a data packet */
    || rcvbuf_xptr->seqno != rb->ph.ackno) {   /* or wrong packet number */
        ++14cnt_discardseq;                    /* count discard */
        14_trace(tr_14discarddat,rb,           /* while waiting for data packets */
              rcvbuf_xptr->seqno);             /* log it */
        12_rcvrelease (rcvbuf_aptr);           /* discard the packet */
        rcvbuf_xptr = NIL;  rcvbuf_rb = NIL;
        }

    else {      /* we can use this packet */

        14_trace(tr_14pktused,rb,rcvbuf_xptr);/* log the use of the packet */
        ++rb->ph.ackno;                        /* accept the sequence number */

        if (rb->rcvptr == rb->bufcursor)       /* if it's the first packet of a msg */
            rb->rcvtype = rcvbuf_xptr->dtype; /* record the type */

        length = rcvbuf_xptr->length - 42;     /* length of user data */
        assert (rb->rcvlength + length <= rb->rcvlimit,
            "14_emptypkt 1");                  /* receive buffer is too small */
        movel (                                /* move the data */
            (faraddr) rcvbuf_xptr + sizeof(struct pkthdr),  /* from */
            (faraddr) (rb->bufcursor),         /* to */
            length);
        rb->bufcursor += length;               /* step to the next position in the buffer */
        rb->rcvlength += length;
        rb->we_owe_ack = TRUE;                 /* we now owe an ack */

        12_rcvrelease (rcvbuf_aptr);           /* release the packet */
        rcvbuf_xptr = NIL;  rcvbuf_rb = NIL;

        if (conctl & eom) {                  /* if end of message */

            if (rb->on_a_list)                     /* if we were on pktwait_list */
               14_removelist (rb, &waitpkt_list); /* we shouldn't be any more */

            if (rb->send_ack) {                    /* if we must send special ack */
```

```
            rb->state = st_sendackc;           /* switch to sending connect ack */
            l4_dosend(rb, l2_getbuf());        /* send the ack */
            }
        else {                                 /* no special ack to send */
            rb->status = l4st_done;            /* so receive is done */
            rb->state = st_idle;               /* and we are idle */
            rb->send_ack = FALSE;              /* no more special ack with eom */
            l4_trace(tr_l4rcvanr,rb,
                rb->ph.srcskt);                /* log rcvmsg done */
            (*rb->anr)(rb);                    /* call his ANR routine */
            }

        } /* eom */

    else if (conctl & ackreq) {            /* ackreq and not end of message */
        rb->state = st_sendackr;           /* switch to sending ack */
        /* Remove the following statement if you want the last packet of a
           multi-packet connection message to generate an ack. */
        rb->send_ack = FALSE;              /* don't need to send special ack */
       /* We must temporarily be taken off the packet-wait list because if there
           are no transmit buffers we will go on the buffer-wait list instead.
           Fear not, l4_dosend will put us back on the packet-wait list. */
        if (rb->on_a_list) l4_removelist (rb, &waitpkt_list);
        l4_dosend(rb, l2_getbuf());        /* send the ack */
        }

    else {                                 /* neither ackreq nor eom */
        rb->state = st_pktwait;            /* so wait for more message packets */
        rb->timer = TO_AWAIT_PKT;
        if (!rb->on_a_list) l4_addlist(rb, &waitpkt_list);
        }

    } /* we can use this packet */

}
```

```
/*---------------------------------------------------------------------------


14_rcvintr (bufptr)          Process receive interrupts.


This is called as an interrupt routine when a packet has been received.
The input parameter "bufptr" points to the arcnet-formatted packet.
We expect that other interrupts from the network device and timer are
disabled.

We call 12_rcvrelease (bufptr) when the received packet can be discarded.
It could be called from this interrupt routine, or later from the timer
interrupt routine or non-interrupt code.  We expect no other calls to
this interrupt routine until the packet is released.

Note that this routine might also be called  when 12_rcvrelease() is
called by anyone else and there is another incoming packet pending.
If it is called from within L4, then the 14_busy flag will be on and
the "interrupt" will be postponed.  That it why code such as
"12_rcvrelease(bufptr);  bufptr = NIL;" doesn't cause a buffer pointer
to the new packet to be destroyed.  In other words, L4 doesn't call itself
recursively as far as incoming packets are concerned.


This routine does the following:

1.  Decode arcnet format and setup the global pointer "rcvbuf_xptr" to
point to the XNS packet embedded within.

2.  Check that it is a valid XNS packet.  Discard it if it is not.

3.  Search the list of RBs waiting for packets, looking for one which
can be given the packet.  There are two cases:

    a.  The packet is a new connection, and the RB is waiting for a
        connection.  Call the ANR routine.

    b.  The packet is part of an existing connection, and the socket
        and connection ids match.  If there is a buffer, move the data.
        If there is no buffer yet, attach the packet and wait.

4.  If no eligible RB is found, setup a timer so that the packet is
discarded if no RB claims it in a short time.


Remember that we are running as an handler from the hardware interrupt
routine, so be discreet!  Make no calls to library routines.
Even "assert" calls that will print a message might be dangerous, but
we allow ourselves that because if the assertion fails the system should
be crashed anyway.

ARCNET note:  The only parts of Level 4 which know about the format of
Arcnet packets are the beginning of 14_rcvintr() and all of 14_fillpkt().
---------------------------------------------------------------------------*/

14_rcvintr (bufptr)

fararcaddr bufptr;  /* pointer to the packet buffer that just arrived */

{
int length;   /* number of data bytes, including XNS header */
int i;
```

```
int conctl;   /* connection control byte for the packet */

struct l4rb *rb;    /* for looking for the rb to assign a packet to */


    if (l4_busy) {

        /* We must postpone this interrupt because l4 is already busy */

        assert (!intpending_rcv, "l4_rcvint 0");
        l4_trace(tr_l4rcvintpost,0,bufptr);   /* log the postponed interrupt */
        intpending_rcv = TRUE;
        intpending_bufp = bufptr; /* save the buffer pointer */
        return;
        }

    assert (!intpending_rcv, "l4_rcvint 1");

    l4_busy = TRUE;


    /* Process the incoming packet interrupt */


    assert (rcvbuf_xptr == NIL, "l4_rcvintr 2");



    /*  Decode short vs. long packet formats and setup packet variables.
        Only this small part of l4_rcvintr is dependent on Arcnet packet
        format.
    */

    rcvbuf_timer = 0;       /* turn off receive packet timeout */
    if (bufptr->count1) {  /* short packet format */
        length = 256 - bufptr->count1;
        rcvbuf_xptr = (farphaddr) ((faraddr) bufptr + bufptr->count1);
        }
    else {  /* long packet format */
        length = 512 - bufptr->count2;
        rcvbuf_xptr = (farphaddr) ((faraddr) bufptr + bufptr->count2);
        }
    rcvbuf_aptr = bufptr;  /* remmber the start of the whole arc packet */


/*---- From here down we are independent of the format of Arcnet packets. ----*/



    l4_trace(tr_l4rcvintr,length,bufptr);   /* log the interrupt */
    ++l4cnt_rcvpkt;                         /* count it */



    /*  Reverse some of the XNS fields if we are running on an (ugh) Intel
        processor.  We only reverse the fields that we do arithmetic on;
        others that are simply compared (srcid, dstid, dsthost, etc.)
        are left as is.

        Whatever fields are reversed here for incoming packets must also be
        reversed in l4_fillpkt() for outgoing packets.
    */

#if intel
/****** We now call an assembly-language routine that does them all at once
        l4_revxns_word (&rcvbuf_xptr->length);
```

```
        14_revxns_word (&rcvbuf_xptr->dstskt);
        14_revxns_word (&rcvbuf_xptr->srcskt);
        14_revxns_word (&rcvbuf_xptr->seqno);
        14_revxns_word (&rcvbuf_xptr->ackno);
        14_revxns_word (&rcvbuf_xptr->allno);
*******/
        12_reverse_xns (rcvbuf_xptr);
#endif


    /*  Check to see if it is a well-formed XNS packet.
        Discard it if not */


    if (
        length & 1                          /* packet size is odd */

    ||  length < 46                         /* datalink packet too small */

    ||  rcvbuf_xptr->sysid != NESTAR_SYSID  /* not Nestar packet type */

    ||  rcvbuf_xptr->length < length-9      /* XNS packet too small */
        /* The maximum discrepancy is 9: 4 from fields not counted by XNS,
           2 from rounding up odd sizes, and 3 from disallowed arcnet sizes. */

    ||  rcvbuf_xptr->length > MAX_XNS_PKT   /* XNS packet too large */

    ||  rcvbuf_xptr->ptype != 5             /* not XNS type SPP = sequence packet protocol */

        /* Should we also check destination the host address? */

    ) {                                     /* bad packet! */
        ++14cnt_discardfmt;                 /* count a discard */
        14_trace(tr_14discardfmt,0,bufptr); /* log a discard due to bad format */
        goto release;
        }


    if (rcvbuf_xptr->dstskt < MAX_WKS) {  /* what kind of socket is it for? */


        /**********  It is a new incoming connection.   ************/


        if (rcvbuf_xptr->seqno != 0             /* seqno and ackno must be zero */
         || rcvbuf_xptr->ackno != 0) {
            ++14cnt_discardcon;                 /* count bad connect */
            14_trace(tr_14discardcon,bufptr,
                    rcvbuf_xptr->seqno);        /* log it */
            goto release;}                      /* discard it */

        if (rcvbuf_xptr->conctl & syspkt) {     /* system packet connect !?! */
            ++14cnt_discardcon;                 /* count bad connect */
            14_trace(tr_14sysconnect,bufptr,
                    rcvbuf_xptr->conctl);        /* log it */
            goto release;                        /* discard it */
            }

        for (i=0; i < MAX_SOCKETS; ++i)       /* check against WKS's we want */
            if (rcvbuf_xptr->dstskt == sockets[i]) goto good_connect;
        14_trace(tr_14badwks,bufptr,
                rcvbuf_xptr->srcskt);         /* log the discard wks connect */
        ++14cnt_badwks;
        goto release;                         /* discard if no wks match */
good_connect:
```

```
    /* Check that this connect packet is not a delayed duplicate
       for a connection already established.
    */

    rb = l4_con_table [hash_addr(rcvbuf_xptr->srchost)];  /* hash into connection table */
    while (rb) {
        if (rb->ph.dstskt == rcvbuf_xptr->srcskt     /* same ephemeral socket? */
         && rb->ph.dstid  == rcvbuf_xptr->srcid) {   /* same source conid? */
            ++l4cnt_discardseq;                      /* yes: discard due to bad sequence */
            l4_trace(tr_l4discarddup,rb,             /* duplicate connect */
                rcvbuf_xptr->seqno);                 /* log it */
            goto release;
            }
        rb = rb->conlink;   /* next rb in this hash list */
        } /* while */


    /* Take the first rb off the waiting-for-connections list, if any.
       Someday we may wish to make selective assignments of connections
       to RBs based on the well-known-socket and the list would be
       searched for an appropriate match.
    */

    if (rb = waitcon_list.head) {           /* somebody is waiting */
        l4_removelist(rb, &waitcon_list);
        assert (rb->state == st_connwait, "l4_rcvintr 3");
        rcvbuf_rb = rb;                     /* grab the packet */
        l4_newconn(rb);                     /* process the connect packet */
        }

    goto set_timer;                         /* wait for an openrcv or rcvmsg */

    } /* new connection */


else {  /* socket test for new connection */

    /*************  It is for a existing connection. ****************

          Search the list of rbs waiting for packets to see if
          it matches anyone.
    */


    rb = l4_con_table [hash_addr(rcvbuf_xptr->srchost)];  /* hash into connection table */

    while (rb) {                                  /* there is an rb in this hash list */

       if (rb->ph.srcskt == rcvbuf_xptr->dstskt   /* if the packet is right socket */
        && rb->ph.srcid  == rcvbuf_xptr->dstid) { /* and right conid */


          /*---------  The packet belongs to this rb. ---------*/


         rcvbuf_rb = rb;                   /* assign us the packet */
         conctl = rcvbuf_xptr->conctl;     /* remember the control byte */

          if (rb->state == st_ackwait) {

              /* We have been waiting for an ack for one of two cases:

                      1.  An outgoing message was completely sent.
```

```
            2.  The first packet of a multiple-packet connect
                message was sent.
    */

    if (rcvbuf_xptr->ackno == rb->ph.seqno) {  /* we got the ack */

        l4_removelist (rb, &waitpkt_list);     /* not waiting any more */

        if (rb->ph.dstskt < MAX_WKS) {    /* have we migrated yet? */
            rb->ph.dstid =                /* no: do so, as follows: */
                rcvbuf_xptr->srcid;       /* incoming src id is our dst id */
            rb->ph.dstskt =
                rcvbuf_xptr->srcskt;      /* incoming src skt is our dst skt */
            rb->ph.maxpkt =
                rcvbuf_xptr->maxpkt;      /* copy MAXPKT_LONG bit.  If off, we
                                             shouldn't send long packets */
            }

        if (conctl & syspkt) {            /* if it is a system packet */
           l2_rcvrelease (rcvbuf_aptr);   /* discard it now */
           /* We discard early so that the rcv buffer isn't tied up during a send */
           rcvbuf_xptr = NIL;  rcvbuf_rb = NIL;
           }

        if (rb->bytes_left == 0)  {      /* the outgoing message is done */
            l4_trace(tr_l4xmitanr,rb,
                rb->ph.srcskt);          /* log the "xmit done" */
            rb->status = l4st_done;      /* the sendmsg is done */
            rb->state = st_idle;
            (*rb->anr)(rb);              /* call his ANR routine */
            }

        else {                           /* multiple packet connect */
            rb->state = st_sending;      /* continue sending */
            l4_dosend (rb, l2_getbuf());
            }

        if (conctl & syspkt) {          /* if it was a system packet */
           goto rti;}                   /* then we're done -- it's discarded */

        if (rcvbuf_xptr->seqno == rb->ph.ackno) { /* if it's data and right seq */
            /* It is also the first data pkt of an incoming message */
            goto set_timer;              /* leave the packet there - wait for a rcvmsg */
            }
        }  /* we got the ack */

    ++l4cnt_discardseq;                 /* discard due to bad sequence */
    l4_trace(tr_l4discardack,rb,        /* while waiting for ack */
        rcvbuf_xptr->seqno);            /* log it */
    goto release;

    }  /* ackwait */


else if (rb->state == st_pktwait) {

    /* We have been waiting for a data packet for a message.
       Note that logic similar to this is in l4_rcvmsg. */

    l4_processpkt (rb);     /* process the packet and release it */

    goto rti;

    } /* pktwait */
```

```
            else goto set_timer;   /* leave packet assigned for a later rcvmsg */

            /* We used to discard an "unexpected" packet here if it is a system packet
               or we are in one of the sending states.  That doesn't work, though,
               because if we are slow compared to the sender he could have sent us the
               packet before we get back to the idle or pktwait state.  So now we leave
               the packet assigned.  If it is truly unexpected, it will be timed out.
               We might still be able to discard unexpected system packets early, but it's
               too hard to think about and will almost never happen, so forget it. */

            }  /* if packet belongs to this rb */

         rb = rb->conlink;                         /* next rb in this hash list */
         }  /* while rb */
      }  /* if existing connection */


   /* At this point either nobody owns up to wanting the packet,
      or the RB who owns it didn't do a rcvmsg yet.
      Set the timer and give somebody a while
      to claim it before it is discarded.

      It can be claimed in any of the following ways:

       a.  By a fresh RB doing an openrcv call.
       b.  By a connected RB doing a rcvmsg call.
       c.  By a connected RB doing some other operation while not
           expecting a packet, and discovering that it owns the
           incoming packet.  It will discard it then, rather than
           waiting for it to timeout.  <--- ?? CHECK THIS.........
   */

set_timer:

   l4_trace(tr_l4unclaimed,rcvbuf_rb,bufptr);    /* log an unclaimed packet */
   rcvbuf_timer = TO_PKT_DISCARD;                /* setup countdown timer */
   goto rti;                                     /* and exit without discarding the packet */



   /*  Release the packet because it has been used or rejected.

       Note that we don't free it as early as we might.
       In particular, we schedule the ack for a message first.
       There's no reason for this;  we could do better but it would
       probably have minimal effect.
   */

release:
   rcvbuf_xptr = NIL;  rcvbuf_rb = NIL;
   l2_rcvrelease (bufptr);  /* discard the packet */


   /* Return from the interrupt */

rti:
   l4_trace(tr_l4rcvintdone,0,bufptr);   /* log the interrupt done */
   l4_busy = FALSE;
   return;
   }
```

```
/*---------------------------------------------------------------------------

14_gotbuf (outbuffer)    Process "got a free buffer" interrupt.

This routine is called by a Level 2 interrupt routine to asynchronously
supply an empty transmit buffer.  The circumstances are as follows:

    1.  A previous call to l2_getbuf returned NIL, indicating that
        there were no free transmit buffers at the time.

    2.  The current interrupt has freed the transmit buffer whose
        address is "outbuffer".

---------------------------------------------------------------------------*/

14_gotbuf(outbuffer)

faraddr outbuffer;      /* the buffer that was just freed */
{
struct 14rb *rb;

    if (14_busy) {

        /* We must postpone this interrupt because 14 is already busy */

        assert(!intpending_xmit,"14_gotbuf 1");
                  /* BUG?  We may have to queue multiple transmit buffers! */

        14_trace(tr_14gotbufpost,0,outbuffer);  /* log the got_buffer interrupt postponed */
        intpending_outbuf = outbuffer;          /* save the buffer pointer */
        intpending_xmit = TRUE;
        return;
        }

    14_busy = TRUE;

    assert (!intpending_xmit, "14_gotbuf 2");


    /* Process the available buffer interrupt */

    rb = waitbuf_list.head;                    /* somebody should be waiting for it */
    assert (rb != NIL, "14_gotbuf 3");
    14_trace(tr_14gotbuf,0,outbuffer);         /* log the got_buffer interrupt processed */
    14_removelist (rb, &waitbuf_list);         /* remove him from the list */
    14_dosend (rb, outbuffer);                 /* let him send */

    14_busy = FALSE;
    }
```

```
/*----------------------------------------------------------------------------


14_timerint ()

Level 4 timer interrupt routine.


This routine is called periodically to process various timeouts.
All the timeout values in the global definition section are in
units which correspond to the frequency with which this routine is
called.  To keep efficiency high the frequency should be low --
something like 4 or 5 per second.

----------------------------------------------------------------------------*/

14_timerint()

{
struct 14rb *rb;  /* for walking list of waiting rb's */

    if (14_busy) {

        /* We must postpone this interrupt because 14 is already busy */

        intpending_timer = TRUE;
        return;
        }

/* We used to:
      assert (!intpending_timer, "14_timerint 1");
   but when debugging with breakpoints or single-step the timer
   can overrun, so don't check.  It doesn't hurt, anyway.
*/

    14_busy = TRUE;


    /*  Possible timeout for incoming packet awaiting processing */


    if (rcvbuf_xptr && rcvbuf_timer > 0) { /* there is a receive packet timer running */

        if (--rcvbuf_timer == 0) {                     /* timer ran out */
            14_trace(tr_14pkttimeout,0,rcvbuf_aptr);  /* log the packet timeout */
            ++14cnt_discardrcv;                        /* count it */
            12_rcvrelease (rcvbuf_aptr);               /* discard the packet */
             rcvbuf_xptr = NIL; rcvbuf_rb = NIL;
            }
        }


    /*  Possible timeout for RBs awaiting incoming packets  */


    for   /* look at each rb waiting for packets */
      (rb = waitpkt_list.head; rb != NIL; rb = rb->flink) {

        assert (rb->timer > 0, "14_timerint 3");
        if (--rb->timer == 0) {  /* timeout awaiting packet */
            14_removelist (rb, &waitpkt_list);  /* remove from waiting list */
```

```
        if (rb->state == st_ackwait) {

            /* Timeout awaiting ack: resend the message */

            if (++rb->snd_count > SEND_RETRIES) {
                l4_doabort(rb);       /* too many message retries: abort the connection */
                ++l4cnt_abortsends;   /* count it */
                (*rb->anr)(rb);       /* and call the ANR */
                }
            else {    /* start up a retry of the previous message */
                ++l4cnt_m_retries;              /* count the message retry */
                l4_trace(tr_l4retry,rb,
                    rb->ph.srcskt);             /* log it */
                rb->bufcursor = rb->sndptr;     /* initialize cursor to start of buffer */
                rb->ph.seqno = rb->first_seq;   /* reset outgoing sequence number */
                rb->ph.allno = rb->ph.seqno + 100; /* arbitrary large allocation number */
                rb->bytes_left = rb->sndlength; /* amount to send */
                if (rb->ph.dstskt < MAX_WKS)    /* we we didn't migrate yet */
                     rb->state = st_connectg;   /* then back to initial connect */
                else rb->state = st_sending;    /* otherwise the sending state */
                l4_dosend(rb, l2_getbuf());     /* process send until blocked */
                }
            }

        else {

            /* Timeout awaiting non-ack message packet: abort the connection */

            assert (rb->state == st_pktwait, "l4_timerint 4");
            l4_trace(tr_l4rcvtimeout,rb,
                rb->ph.srcskt);             /* log the rcv timeout */
            ++l4cnt_abortrcvs;              /* count it */
            l4_doabort(rb);
            (*rb->anr)(rb);                 /* call ANR */
            }

            break;  /* if we found one timeout, don't look for others
                       because the list has changed.  Catch them the next time.
                    */
        } /* timeout found */

    } /* for rb */

l4_busy = FALSE;
}
```

```
/*---------------------------------------------------------------------------

14_fillpkt (&14rb, &buffer, ackrequest, systempkt)


Fill the transmit packet buffer with data from the message and
queue the packet for transmission.


Input:  "buffer" points to the beginning of the arcnet-format packet,
        that is, the 1-byte SID field.

        "ackrequest" is TRUE if we should request an ack even if this
        isn't the last packet of the message.  This is used to demand
        an ack after the first packet of a multiple-packet initial
        connect message.

        "systempkt" is TRUE if we are sending a system packet ack with
        no data.  Note that this is different from a zero-length
        data packet.

        rb->bytes_left  is the number of bytes left to send in this message.

        rb->bufcursor  is the pointer to the data to send.


If this is the last packet of the message, set the ack-request and eom bits on.

All packets are currently formatted as if they were RIM buffer arcnet
packets, including the empty space.  For token ring, empty space can be
omitted if both sides agree;  only this routine would need to change
for transmission, and 14_rcvintr() for reception.


ARCNET note:  The only parts of Level 4 which know about the format of
Arcnet packets are the beginning of 14_rcvintr() and all of 14_fillpkt().

---------------------------------------------------------------------------*/

14_fillpkt (rb, buffer, ackrequest, systempkt)

struct 14rb *rb;
fararcaddr buffer;
boolean ackrequest, systempkt;

{ short int bytes_to_do;  /* bytes to send, including xns header */
  short int buf_offset;   /* where in RIM buffer to start the data */
  farphaddr xnshdr;       /* pointer to XNS header in the buffer */
  boolean longpktok;      /* are long packets ok? */


  longpktok = rb->ph.maxpkt & MAXPKT_LONG;  /* long packets ok? */


    /*  Fill in the arcnet destination address from the fifth byte
        of the XNS destination host address.  The arcnet source
        address is supplied by the hardware.
    */
    buffer->did = *( (addr) rb->ph.dsthost + 5 );  /* (works for Moto OR Intel!) */


    rb->we_owe_ack = FALSE;  /* we will be sending an implicit ack */
```

```
bytes_to_do = sizeof(struct pkthdr);            /* size of xns header */
if (!systempkt) bytes_to_do += rb->bytes_left;  /* plus data, maybe */


/*  Setup various Arcnet packet formats depending on the size
    of the header + data to be sent.
*/


if (bytes_to_do <= 252) {

    /* Case 1: This is the last packet of the message and is a short
       packet or is a system-packet ack. */

    if (systempkt) {
        rb->ph.conctl = syspkt;  /* system-packet ack */
        rb->ph.dtype = 0;        /* zero packet type for sniffer neatness */
        }
    else {
        rb->ph.conctl = ackreq + eom;  /* ack request and end-of-message */
        }

    buf_offset = 256 - bytes_to_do;    /* start of data in pkt buffer */
    if (buf_offset & 1) {              /* can't be odd */
        --buf_offset;
        rb->ph.garbage = 1;
        }
    else rb->ph.garbage = 0;

    buffer->count1 = buf_offset;       /* short continuation ptr */
    rb->ph.length = bytes_to_do - 4;
    movel ( (faraddr) (&rb->ph), /* from */
        (faraddr) buffer + buf_offset, /* to */
        sizeof(struct pkthdr));  /* move the xns header */
    if (!systempkt && rb->bytes_left) {  /* move the data, if any */
        movel ( (faraddr) (rb->bufcursor),  /* from */
            (faraddr) buffer + buf_offset + sizeof(struct pkthdr),  /* to */
            rb->bytes_left);  /* length */
        rb->bytes_left = 0;
        }
    }


else if (bytes_to_do <= 508  &&  longpktok) {

    /* Case 2: This is the last packet of the message, and is long, and
       we are allowed to send long packets, so do so. */

    rb->ph.conctl = ackreq + eom;       /* ack request and end-of-message */
    buffer->count1 = 0;                 /* flag indication long packet */

    if (bytes_to_do < 258) {            /* 253..257 must be sent as 258 */
        rb->ph.garbage = 258 - bytes_to_do;  /* number of pad bytes */
        buf_offset = 254;                  /* start of data in pkt buffer */
       }
    else {                                 /* 258..508 */
        buf_offset = 512 - bytes_to_do;    /* start of data in pkt buffer */
        if (buf_offset & 1) {              /* can't be odd */
            --buf_offset;
            rb->ph.garbage = 1;
            }
        else rb->ph.garbage = 0;
        }
```

```
        buffer->count2 = buf_offset;        /* long continuation ptr */
        rb->ph.length = bytes_to_do - 4;
        movel ( (faraddr) (&rb->ph),        /* from */
            (faraddr) buffer + buf_offset, /* to */
            sizeof(struct pkthdr));        /* move the xns header */
        movel ( (faraddr) (rb->bufcursor), /* from */
            (faraddr) buffer + buf_offset + sizeof(struct pkthdr), /* to */
            rb->bytes_left);               /* move the data */
        rb->bytes_left = 0;
        }


    else if (longpktok) {

        /* Case 3: This is not the last packet of the message, and we are
           allowed to send long packets, so send a long packet. */

        bytes_to_do = 508 - sizeof(struct pkthdr);  /* # of data bytes to send */
        buffer->count1 = 0;           /* flag indicating long packet */
        buffer->count2 = 4;           /* long continuation ptr for max pkt */
        rb->ph.length = 508-4;
        buf_offset = 4;
sendfull:
        if (ackrequest)
             rb->ph.conctl = ackreq;    /* force an ack request */
        else rb->ph.conctl = 0;         /* otherwise no connection control bits */
        rb->ph.garbage = 0;             /* no pad bytes */
        movel ( (faraddr) (&rb->ph),    /* from */
            (faraddr) buffer + buf_offset, /* to */
            sizeof(struct pkthdr));     /* move the xns header */
        movel ( (faraddr) (rb->bufcursor), /* from */
           (faraddr) buffer + buf_offset + sizeof(struct pkthdr), /* to */
           bytes_to_do);                /* move the data */
        rb->bytes_left -= bytes_to_do;  /* decrement count by amount sent */
        rb->bufcursor  += bytes_to_do;  /* increment pointer to data by amount sent */
        }


    else {

        /* Case 4: This is not the last packet of the message, but we are
           not allowed to send long packets, so send short. */

        bytes_to_do = 252 - sizeof(struct pkthdr);  /* # of data bytes to send */
        buffer->count1 = 4;           /* short continuation pointer for max pkt */
        rb->ph.length = 252-4;
        buf_offset = 4;
        goto sendfull;
        }


    /*  Reverse some of the XNS fields if we are running on an (ugh) Intel
        processor.  We only reverse the fields that we do arithmetic on;
        others that are simply compared (srcid, dstid, dsthost, etc.)
        are left as is.

        Whatever fields are reversed here for outgoing packets must also be
        reversed in l4_rcvintr() for incoming packets.
    */

#if intel
/****** We now call an assembly-language routine that does them all at once
        xnshdr = (farphaddr) ( (faraddr) buffer + buf_offset);
        l4_revxns_word (&xnshdr->length);
```

```
        14_revxns_word (&xnshdr->dstskt);
        14_revxns_word (&xnshdr->srcskt);
        14_revxns_word (&xnshdr->seqno);
        14_revxns_word (&xnshdr->ackno);
        14_revxns_word (&xnshdr->allno);
*******/
        12_reverse_xns ( (faraddr) buffer + buf_offset);
#endif


    /*    Queue the packet for transmission and return. */

    12_sendbuf(buffer);                 /* send the buffer we just filled */
    ++14cnt_sendpkt;                    /* count it */
    14_trace(tr_14pktsent,rb,buffer); /* log it */
    /*...14_trace(tr_14pktsize,xnshdr->length,xnshdr->seqno); /* log the size and seqno */
    if (!(rb->ph.conctl & syspkt))      /* if it's not a system packet */
        ++rb->ph.seqno;                 /* then increment the packet sequence number for next time */
    }
```

```
/*--------------------------------------------------------------------------*/

    l4_addlist (rb, list)

/*  Add an rb to end of one of the waiting lists.

    We add to the end so that the "waitbuf" list will work like a
    FIFO queue.  For the other lists it doesn't matter.
*/

struct l4rb *rb;        /* the rb to add */
struct l4_list *list;   /* the list to add it to */

/*--------------------------------------------------------------------------*/

{
    assert (!rb->on_a_list, "l4_addlist 1");  /* better not be on a list already! */

    rb->flink = NIL;                        /* no forward link */
    rb->blink = list->tail;                 /* back link is to previous tail */
    if (list->tail)                         /* if there was a previous tail */
        list->tail->flink = rb;             /*    it points to us */
    list->tail = rb;                        /* we are now the tail */
    if (!list->head) list->head = rb;       /* if we are the head too */

    rb->on_a_list = TRUE;
    }



/*--------------------------------------------------------------------------*/

    l4_removelist (rb, list)

/*  Remove an rb from one of the waiting lists.  */

struct l4rb *rb;        /* the rb to add */
struct l4_list *list;   /* the list to add it to */

/*--------------------------------------------------------------------------*/

{
    assert (rb->on_a_list, "l4_removelist 1");

    if (rb == list->head)                   /* if we are the head */
        list->head = rb->flink;             /* then the new head is our next */
    else rb->blink->flink = rb->flink;      /* otherwise make our previous point to our next */

    if (rb == list->tail)                   /* if we are the tail */
        list->tail = rb->blink;             /* then the new tail is our previous */
    else rb->flink->blink = rb->blink;      /* otherwise make our next point to our prevoius */

    rb->flink = NIL;                        /* for neatness only */
    rb->blink = NIL;                        /* for neatness only */

    rb->on_a_list = FALSE;
    }
```

```
/*---------------------------------------------------------------------------*/

    14_purgelist (srb, list)

/* Purge an rb from a list if it is there */

struct 14rb *srb;       /* the rb to purge */
struct 14_list *list;  /* the list it might be on */

/*---------------------------------------------------------------------------*/

{
struct 14rb *rb;

    for   /* search the list for the rb */
     (rb = list->head; rb != NIL; rb = rb->flink) {
        if (rb == srb) {
            14_removelist(rb, list);
            break;}
        } /* for */
    }



/*---------------------------------------------------------------------------*/

    14_removecon (srb)

/*  Remove an rb from the connection hash table.

    This is called from places that terminate the connection:

        14_doabort   for aborts
        14_disconn   for disconnects when no ack is due
        14_dosend    for disconnects when an ack was due
*/

struct 14rb *srb;       /* the rb to remove */

/*---------------------------------------------------------------------------*/

{
short int hash_index;
register struct 14rb *rb, *prevrb;

    hash_index = hash_addr(srb->ph.dsthost);       /* hash the address into an index */

    if (14_con_table [hash_index] == srb) {        /* the 99% case: it is the head */
        14_con_table [hash_index] = srb->conlink;  /* so remove it */
        return;                                    /* and return */
        }

    prevrb = 14_con_table [hash_index];        /* otherwise search the list */
    while (rb = prevrb->conlink) {
        if (rb == srb) {                       /* found */
            prevrb->conlink = rb->conlink;     /* unlink it */
            return;
            }
        prevrb = rb;
```

```
    } /* while */
assert (FALSE,"14_removecon 1");    /* rb not found in connect table */
}
```

```
#if intel

/*---------------------------------------------------------------------------*/

        14_revxns_word ( ptr )

/*  Reverse the word whose address is in the long pointer "ptr".
    This is used for reversing msb-first fields in the XNS header
    if we are unfortunate enough to be executing on an Intel processor.

    The reversal is done in place in the Arcnet RIM buffer, hence
    the long pointer.

*/

faraddr ptr;   /* a long pointer to a character */

/*---------------------------------------------------------------------------*/

{   byte temp;

    temp = *ptr;
    *ptr = *(ptr+1);
    *(ptr+1) = temp;
}

#endif  /* intel */
```

```
/*----------------------------------------------------------------------

            Remaining notes and questions

----------------------------------------------------------------------

2.  Should routines that complete immediately set the ending status and
not call the ANR routine?  What about the case where the status changes
from busy to complete after returning but before the caller tests the
status?  In that case the ANR routine will be called gratuitously if
the caller notices the status instead of waiting for the ANR.  The
easy solution is to recommend that the caller ALWAYS wait for the ANR
before checking the status.

6.  To add arcnet/token ring simultaneous support, we need to:

    a.  Pass and receive the rb->arcnet boolean to the Level 2 routines.

    b.  Generalize l4_gotbuf() so that it searches for the first rb
        waiting for the appropriate kind of buffer rather than just
        using the first rb on the list.  Or, better, have two waiting
        lists for empty buffers.

    c.  Add a new l4_fillpkt routine for token ring, and modify the first
        part of l4_rcvintr.

    d.  Write the token ring Level 2 packet routines.


7. Need to add broadcast support for both incoming and outgoing
   connections.

8. Add l4_acknow() function?

*/

/* end of l4.c */
```