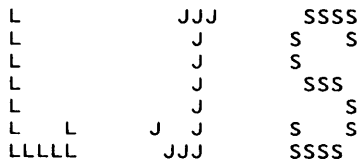


Wed 19-Mar-1986 15:23:04

Print request number 488

Station: $36

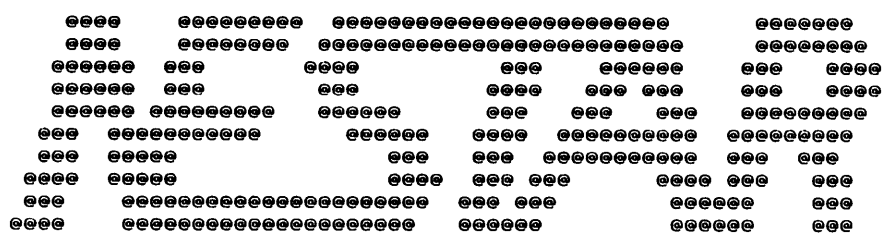
Name: L J Shustek

File Server: BUTLER ($FE)

NFS Pathname:

Filename (s):

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



```
/* this is l4test2.c */

/*****************************************************************************

        Test program for interrupt-driven Level 4


This engages two stations in a multiple-connection dialog to test the
interrupt-driven Level 4.

Any number of pairs of connections can be established.  Each pair
has a conversation as follows:

        stn 1               stn 2
        --------------      -------------

        connnect            open_rcv, rcv_msg

        rcv_msg             send_msg

        send_msg            rcv_msg

        disconnect          disconnect


The number of connections and message sizes can be controlled by
the test operator.


*****************************************************************************/

/* --------  Change log  --------

 1/10/86    L. Shustek    Written for pc Level 4 testing.
 2/ 4/86    L. Shustek    Add repeat loop.  Reorder counter display.
 2/ 5/86    L. Shustek    Prompt for number of rb's and connections.
 2/11/86    L. Shustek    Add display of final rb status.
                          Add message data and datastream type checks.
                          Fix trace dump wrap problem.
 2/12/86    L. Shustek    Change to FIFO processing of RBs because the
                          unfairness was causing timeouts.
 3/13/86    L. Shustek    Replace scanf() with our own getnum() function.
                          Other minor changes for port to 68000.
*/

#include "exec.h"
#include "l4.h"
#include "l4counts.h"

#if microtec
#include "ctype68.h"
#else
#include "ctype.h"
#endif

#if microsoft
#include <stdio.h>
#include <conio.h>
#endif

#define MAXRBS 10       /* maximum number of rbs (connections) */
#define BUFMAX 4096     /* maximum buffer size (must be power of 2) */
#define SKT 0x40C       /* the well-known socket we use */
```

```
#define MTRC  200      /* the size of the trace buffer */
#define STOP_ON_ERROR  FALSE  /* should we stop if a connection is aborted? */
```

```
/*
    Define the Level 4 request blocks.

    In addition, we define an "RB extension" where we store private
    state information to indicate what is to be done next for
    the connection.

*/

struct l4rb rbs[MAXRBS];    /* all the rb's we use */

struct l4rbx {              /* rb extension;  one for each rb */
    enum {connect1, connect2, connect3, connect4, connect5, connect6,
          receive1, receive2, receive3, receive4, receive5, receive6}
        state;              /* state of this rb */
        int cycles;                         /* number of connection cycles run */
    char buffer [BUFMAX];  /* buffer for this rb */
    }rbxs[MAXRBS];

struct l4rb *active_list;   /* the head of a list of rb's needing attention */
struct l4rb *active_tail;   /* the tail of the list of rb's needing attention */


char *l4_status_names[] = {  /* l4 rb status */
    "uncon", "busy", "done",
    "???", "???", "???"};

char *l4_state_names[] = {  /* l4 rb state */
    "idle",    "connectg", "sending",  "sendackr", "sendackc", "sendackd",
    "pktwait", "ackwait",  "connwait", "???",      "???",      "???"};

char *rbx_state_names[] = {  /* rbx state */
        "connect1", "connect2", "connect3", "connect4", "connect5", "connect6",
        "receive1", "receive2", "receive3", "receive4", "receive5", "receive6",
        "???",      "???",      "???"};


/*
          Other global variables
*/


int his_addr;             /* his arcnet address */
int msg_size;                 /* size of outgoing messages */
int nrbs;                             /* the number of rbs (connections) */
int noutgoing;            /* the number of outgoing connections */

int aborts;               /* number of aborted connections */

int msg_count = 0;        /* outgoing message counter */
long int trace_seq = 0;   /* count of trace entries */

#if intel  /* really "if pc"  */
typedef int far *farintaddr;
farintaddr timer = (farintaddr)0x0040006c;  /* location of timer on pc/xt/at */
#endif
```

```
/*              Trace entry names.

                These must match the defintions of tr_l4xxxxxxx macros in
                the file debug.h.

                The ones marked with dashes represent error or unusual conditions.
*/

#define  tr_l4connect 50        /* the number of the first l4 trace entry type */

char *l4_trace_names[] = {

"l4connect",      "l4openrcv",       "l4sendmsg",       "l4rcvmsg",        "l4disconn",
"-----l4abort",    "l4rcvanr",       "l4xmitanr",       "l4connectrcvd",   "l4disconned",
"l4rcvintr",      "---l4discardfmt","---l4discardcon","---l4discardack","---l4sysconnect",
"----l4badwks",
"--l4pkttimeout", "--l4rcvtimeout","-----l4retry",    "l4unclaimed",     "l4pktused",
"l4pktsent",      "l4gotbuf",        "l4gotbufpost",    "---l4discarddat","l4rcvintpost",
"l4rcvintdone",   "l4pktsize",       "---l4discarddup","---l4discardunx","???",
"???"};


struct {                        /* trace buffer */
    long int seq;
        int time;
        int code;
        int info1,info2;
        } trace_buf[NTRC];

int trace_ptr = 0;   /* next free location in trace buffer */


/*
    Event trace routine.

    Store the events in a circular buffer for later display.

*/

do_trace (code, info1)   /* alternate form with only 1 info parm */
int code;
lword info1;
{ do_l4_trace (code, (int) info1, (int) 0); }


do_l4_trace (code, info1, info2)

int code;
int info1, info2;
{
    /* printf("%x %s\n",rb,l4_trace_names[code-tr_l4connect]); */
        trace_buf[trace_ptr].seq = ++trace_seq;
        trace_buf[trace_ptr].code = code;
#if intel
        trace_buf[trace_ptr].time = *timer;  /* pc only! */
#else
        trace_buf[trace_ptr].time = 0;  /* temporary, until we have timer */
#endif
        trace_buf[trace_ptr].info1 = info1;
        trace_buf[trace_ptr].info2 = info2;
        if (++trace_ptr >= NTRC) trace_ptr = 0;
#if 0
/*TEMP*/  if (code == 13 /*tr_l4discardack*/ || code == 24 /*tr_l4discarddat*/) {
                        printf("Aborting due to discard\n");
```

```
                                l4_terminate();
                                trace_dump();
                                count_dump();
                                exit(97);
/*TEMP*/                    }
#endif
    }


/*
    Dump the trace buffer and rb status.

*/

trace_dump()                        /* print the trace table */

{
int i;
int oldtime, delta;

        i=trace_ptr;
        oldtime = 0;
    do {
                if (trace_buf[i].code >= 0 ) {
                    delta = trace_buf[i].time - oldtime;
                    oldtime = trace_buf[i].time;
                        printf("%6ld %6u %16s  %4x  %4x\n",
                                trace_buf[i].seq,
                                delta,
                                l4_trace_names[trace_buf[i].code-tr_l4connect],
                                trace_buf[i].info1,
                                trace_buf[i].info2);
                        }
                ++i; if (i >= NTRC) i = 0;
                }  while (i!=trace_ptr && !kbhit());

        for (i=0; i<nrbs; ++i) {
                printf("RB %d at %x, status %s, state %s, rbx state %s, cycles %d\n",
                        i, &rbs[i],
                        l4_status_names[rbs[i].status],
                        l4_state_names[rbs[i].state],
                        rbx_state_names[(int)(rbxs[i].state)],
                        rbxs[i].cycles);
        printf("  length = $%x, dtype = %d, conctl = $%x\n",
            rbs[i].ph.length, rbs[i].ph.dtype, rbs[i].ph.conctl);
        printf("  srcskt = $%x, dstskt = $%x, srcid = $%x, dstid = $%x, seq = %d, ack = %d\n",
            rbs[i].ph.srcskt, rbs[i].ph.dstskt, rbs[i].ph.srcid, rbs[i].ph.dstid,
            rbs[i].ph.seqno, rbs[i].ph.ackno);
                } /* for */
        }


count_dump()                        /* print the L4 counters */
{
printf("%5u outgoing connections      %5u incoming connections\n",l4cnt_connect,l4cnt_openrcv);
printf("%5u messages sent             %5u messages received    \n",l4cnt_sendmsg,l4cnt_rcvmsg);
printf("%5u packets sent              %5u packets received     \n",l4cnt_sendpkt,l4cnt_rcvpkt);
printf("%5u transmit interrupts       %5u receive interrupts   \n",l4cnt_l2int_ta,l4cnt_l2int_ri);
printf("%5u send msg retries          %5u send pkt retries     \n",l4cnt_m_retries,l4cnt_p_retries);
printf("%5u transmit without ack      %5u transmit timeouts    \n",l4cnt_xmitnoack, l4cnt_xmittimeout);
printf("%5u bad format discards       %5u bad connect discards\n",l4cnt_discardfmt,l4cnt_discardcon);
printf("%5u bad sequence discards     %5u bad socket discards \n",l4cnt_discardseq,l4cnt_badwks);
printf("%5u receive timeout discards  %5u unxpctd pkt discards\n",l4cnt_discardrcv,l4cnt_discardunx);
printf("%5u network reconfigurations  %5u connection aborts    \n",l4cnt_l2int_recon,l4cnt_aborts);
printf("%5u send retry aborts         %5u rcv timeout aborts   \n",l4cnt_abortsends,l4cnt_abortrcvs);
```

```
printf("%5u aborts returned to 14test\n",aborts);
        }
```

```
/*****************************************************************************

                    MISCELLANEOUS ROUTINES

*****************************************************************************/


zero (ptr, size)                /* zero memory */
addr ptr; int size;
{
        while (size-- > 0) *ptr++ = 0;
        }



anr_rtn (rb)    /* Our Asynchronous Notification Routine */
struct l4rb *rb;
{
short int flags;                        /* saved enable/disable state */

    if (rb->status == l4st_busy) panic("anr 1");
    flags = disable();      /* hold interrupts */
        /* Add this rb to the end of the active list so processing is FIFO. */
        if (active_list) active_tail->flink = rb;
                else active_list = rb;
        active_tail = rb;
                rb->flink = NIL;
    enable(flags);                      /* resume interrupts */
        }



/* Internal error.

   This gets called by panic() to handle internal errors.
   For the 8088 panic() is in machine code to reset
   the stack to something C runtime routines will be happy with.
*/


    panic2 (msg)
    char *msg;
    {
            printf("Assertion failed: %s\n",msg);
            l4_terminate();
            trace_dump();
            count_dump();
            printf("Assertion failed: %s\n",msg);
            }

#if microtec
    panic (msg)
    char *msg;
    { panic2(msg); }
#endif


/*
                Make up a message to send.
*/

#if microtec
    int _randx = 0;
#endif
```

```
make_msg(rb, code)

struct l4rb *rb;
int code;
{
struct l4rbx *rbx;
int i,length;

    rbx = (struct l4rbx *) rb->user;
        rb->sndptr = rbx->buffer;
        length = msg_size ? msg_size
                          : (rand() & BUFMAX-1) + 1;
    rb->sndlength = length;
    rb->sndtype = code;
        /* Fill in every 100th byte with values based on the
      length of the message, for the receiver to check.
           This is much quicker than filling the whole buffer, but checks
           that all the packets were received in the right order.
           Code in check_msg looks for what we did here.
        */
    for (i=0; i < rb->sndlength; i += 100) {
                rbx->buffer[i] = length++;  /* truncates to 0..255 */
                }
    }



/*
        Check the received message.

        Check the datastream type and some of the data.
        If it's ok, return TRUE.
        If something's wrong, print a message and return FALSE.
*/

boolean check_msg(rb, code)

struct l4rb *rb;
int code;
{
struct l4rbx *rbx;
int i,length;

    rbx = (struct l4rbx *) rb->user;

        if (rb->rcvtype != code) {  /* check datastream type */
                printf("Wrong type, rb %x, expected $%x, got $%x\n",
                    rb, code, rb->rcvtype);
                return FALSE;
                }

    length = rb->rcvlength;
    for (i=0; i < rb->rcvlength; i += 100) {  /* check data */
                if (rbx->buffer[i] != (char)length++) {
                        printf("Wrong data, rb %x, offset %d\n", rb, i);
                    return FALSE;
                        }
                } /* for */
    return TRUE;
    }
```

```
/*****************************************************************************

                    INITIALIZATION

*****************************************************************************/


init()                  /* Initialization */
{
struct l4rb *rb;
struct l4rbx *rbx;
int i;

    active_list = active_tail = NIL;
    for (i=0; i<nrbs; ++i) {            /* for all connections */
        rb = &rbs[i];
        rbx = &rbxs[i];
            zero (rb, sizeof(struct l4rb));   /* initialize rb and rbx */
        zero (rbx, sizeof(struct l4rbx));
        rb->anr = anr_rtn;                /* point rb to anr routine */
        rb->user = (addr) rbx;            /* point rb to rbx */
        rb->id = RBid;
                rbx->cycles = 0;
        if (i < noutgoing) {        /* Of all the connections, */
                        rbx->state = connect1; /* some are connecting */
                                                        /* but are quiescent */
                        }
        else {rbx->state = receive1; /* the rest are receiving */
                        anr_rtn(rb);                    /* and are active */
                        }
                } /* for */

     aborts = 0;

     l4cnt_connect      = 0;    /* outgoing connections */
     l4cnt_openrcv      = 0;    /* incoming connections */
     l4cnt_sendmsg      = 0;    /* messages sent */
     l4cnt_rcvmsg       = 0;    /* messages received */
     l4cnt_sendpkt      = 0;    /* packets sent */
     l4cnt_rcvpkt       = 0;    /* packets received */
     l4cnt_discardfmt   = 0;    /* bad format packets discarded */
     l4cnt_discardcon   = 0;    /* bad connect packet discarded */
     l4cnt_discardseq   = 0;    /* bad sequence packets discarded */
     l4cnt_discardunx   = 0;    /* unexpected packet discard */
     l4cnt_discardrcv   = 0;    /* rcv packet timeout discard */
     l4cnt_badwks       = 0;    /* incoming connects on wrong wks */
     l4cnt_m_retries    = 0;    /* send retry attempts */
     l4cnt_p_retries    = 0;    /* send retry attempts */
     l4cnt_aborts       = 0;    /* connections aborts */
     l4cnt_abortsends   = 0;    /* connection aborts due to send retries w/o ack */
     l4cnt_abortrcvs    = 0;    /* connection aborts due to rcv timeouts */
     l4cnt_l2int_ri     = 0;    /* Level 2 receive interrupts */
     l4cnt_l2int_ta     = 0;    /* Level 2 transmit interrupts */
     l4cnt_l2int_recon  = 0;    /* Level 2 recon interrupts */
     l4cnt_xmittimeout  = 0;    /* Level 2 transmit timeouts */
     l4cnt_xmitnoack    = 0;    /* Level 2 transmit w/o ack (TA w/o TMA) */
    }
```

```
/*****************************************************************************

                RB STATE MACHINE

*****************************************************************************/

/*  Move an rb to the next state in the process of the connection.
    Return TRUE if processing was ok.
    Return FALSE if the connection aborted.
*/

boolean process(rb)

struct l4rb *rb;
{
struct l4rbx *rbx;

    rbx = (struct l4rbx *) rb->user;

    switch (rbx->state) {


/* The state machine for an rb which is the connection initiator */


        case connect1:
                panic("connect1 state in l4test process!");

connect:

        ++rbx->cycles;

    case connect2:          /* send a connect message */

        make_msg(rb,10);
        rb->ph.dstskt = SKT;
#if intel
        rb->ph.dsthost[2] = his_addr << 8;
#else
        rb->ph.dsthost[2] = his_addr;
#endif
        rb->arcnet = TRUE;
        rbx->state = connect3;
        l4_connect(rb);
        break;

    case connect3:          /* receive a response message */

        if (rb->status != l4st_done)
            {++aborts; if (STOP_ON_ERROR) goto tabort; else goto connect;}
        rb->rcvptr = rbx->buffer;
        rb->rcvlimit = BUFMAX;
        rbx->state = connect4;
        l4_rcvmsg(rb);
        break;

    case connect4:      /* generate a final message */

        if (rb->status != l4st_done)
            {++aborts; if (STOP_ON_ERROR) goto tabort; else goto connect;}
            if (!check_msg(rb,11)) goto tabort;
        make_msg(rb,12);
        rbx->state = connect5;
```

```
        l4_sendmsg(rb);
        break;

    case connect5:          /* disconnect */

        if (rb->status != l4st_done)
            {++aborts; if (STOP_ON_ERROR) goto tabort; else goto connect;}
        rbx->state = connect6;
        l4_disconn(rb);
        break;

    case connect6:          /* check disconnect, and go connect again */

        if (rb->status != l4st_uncon)
            {++aborts; if (STOP_ON_ERROR) goto tabort; else goto connect;}
        goto connect;


/*  The state machine for an rb which is a connection receiver */

receive:

        ++rbx->cycles;

    case receive1:          /* receive a connection */

                rbx->state = receive2;
        l4_openrcv(rb);
        break;

    case receive2:          /* receive the message */

        if (rb->status != l4st_done)
            {++aborts; if (STOP_ON_ERROR) goto tabort; else goto receive;}
            if (!check_msg(rb,10)) goto tabort;
        rb->rcvptr = rbx->buffer;
        rb->rcvlimit = BUFMAX;
        rbx->state = receive3;
        l4_rcvmsg(rb);
        break;

    case receive3:         /* generate a response */

        if (rb->status != l4st_done)
            {++aborts; if (STOP_ON_ERROR) goto tabort; else goto connect;}
        make_msg(rb,11);
        rbx->state = receive4;
        l4_sendmsg(rb);
        break;

    case receive4:          /* receive the final message */

        if (rb->status != l4st_done)
            {++aborts; if (STOP_ON_ERROR) goto tabort; else goto connect;}
        rb->rcvptr = rbx->buffer;
        rb->rcvlimit = BUFMAX;
        rbx->state = receive5;
        l4_rcvmsg(rb);
        break;

    case receive5:          /* disconnect */

        if (rb->status != l4st_done)
            {++aborts; if (STOP_ON_ERROR) goto tabort; else goto connect;}
            if (!check_msg(rb,12)) goto tabort;
```

```
        rbx->state = receive6;
        l4_disconn(rb);
        break;

    case receive6:          /* check disconnect, and go receive again */

        if (rb->status != l4st_uncon)
            {++aborts; if (STOP_ON_ERROR) goto tabort; else goto connect;}
        goto receive;


        } /* switch */

    return TRUE;    /* everything ok */

tabort:
    printf("Status for rb %x: %d (%s). Test terminated.\n",
            rb, rb->status, l4_status_names[rb->status]);
    return FALSE;
    }
```

```
/*************************************************************************

                Simple-minded Utilities

*************************************************************************/


/*  Get a number from the keyboard, in base 10 or 16 as specified.
    Skip leading blanks.  Return FALSE if no number found.
*/


boolean getnum ( num, base )
int *num, base;
{
char ch;
boolean gotit = FALSE;

    *num = 0;
    while ( (ch=getch()) != '\n'  &&  ch!='\r') {
        ch = toupper(ch);
        if (ch == ' ' &&  !gotit) continue;
        gotit = TRUE;
        if (isdigit(ch)) *num = *num * base  +  (ch-'0');
        else if (base==16 && isxdigit(ch))
            *num = *num * base + (ch-'A'+10);
        else return FALSE;
        }
    return gotit;
    }


#if microtec
getch()
{char ch;
  ch = putchar(getchar());
  if (ch == '\r') putchar('\n');
  return ch;}
#endif
```

```
/*************************************************************************

                    THE MAIN DRIVER

*************************************************************************/

main()
{
int i;
char ch;
struct l4rb *rb;
short int flags;                    /* saved enable/disable state */

printf("Test pgm for interrupt-driven L4\n");

printf("Other station's arcnet address: $");
while (!getnum(&his_addr,16)
    || his_addr<1 || his_addr>254) printf("Error. Try again\n");

srand(his_addr);  /* seed the random number generator */

while (TRUE) {  /* repeat for all tests */

    printf("Number of rbs (1..%d): ",MAXRBS);
        while (!getnum(&nrbs,10)
          || nrbs<1 || nrbs>MAXRBS) printf("Error. Try again\n");

    printf("Number of outbound connections (0..%d): ",nrbs);
        while (!getnum(&noutgoing,10)
          || noutgoing<0 || noutgoing>nrbs) printf("Error. Try again\n");

        printf("Message size in bytes (0 for random, or 1..%d): ",BUFMAX);
        while (!getnum(&msg_size,10)
      || msg_size<0 || msg_size>BUFMAX) printf("Error. Try again\n");

    init();     /* initialize rbs and counters */

        for (i=0;  i < NTRC; ++i)   /* intialize trace table */
            trace_buf[i].code = -1;

    if (!l4_init()) {
        printf("Initialization failed\n");
        return;}

        l4_listen (SKT);

        printf("Press  ESC    to abort with trace dump.\n");
        printf("Press  SPACE  to stop gracefully.\n");
        printf("Press  C      to start outbound connections.\n");

    while (TRUE) {  /* wait for interesting things to happen */

                /*  Process keyboard commands */

                if (kbhit()) {   /* got a key */
                        ch = getch();

                        if (ch == 0x1b  ||  ch == ' ') {  /* ESC or space: abort */
abort:          printf("Aborting...\n");
                            l4_unlisten(SKT);
                               l4_terminate();
                               if (ch == 0x1b) trace_dump();  /* dump only if ESC */
                               count_dump();
                               goto next_test;
```

```
                        }

                if (ch == 'C') {  /* C: start connections */
                        for (i=0; i<nrbs; ++i) {  /* look for quiescent rbs */
                                if (rbxs[i].state == connect1) {  /* who want to connect */
                                        rbxs[i].state = connect2;  /* and start them */
                                        anr_rtn(&rbs[i]);
                                        }
                                } /* for */
                        }

                } /* kbhit */


        /*  Process any rbs put on the active list by the ANR routine. */


    if (active_list) {                /* if there is an rb to process */
        flags = disable();
            rb = active_list;         /* remove it from the active list */
            active_list = rb->flink; /* (while interrupts are disabled) */
            rb->flink = NIL;
        enable(flags);
        if (!process(rb)){            /* and process it */
                        ch = 0x1b;                          /* simulate ESC if connection aborted */
                        goto abort;
                        }
        }

    } /* while TRUE wait */


next_test:;
    } /* while TRUE repeat for all tests */
}  /* main */

/* end of l4test2.c */
```

```
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
```

Fri 24-Jan-1986 11:57:30

Print request number 14

Station: $36

Name: L J Shustek

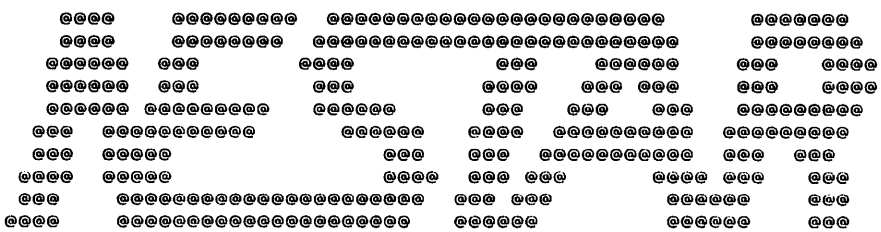
File Server: BUTLER ($FE)

NFS Pathname:

Filename (s):

L4TEST (old; want to be updated!)

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



```
/* this is l4test.c */


/*
        Test program for interrupt-driven Level 4

                Receive a text message and respond in kind.
*/

#include <\fs\exec\exec.h>
#include <l4.h>
#include <o:\include\stdio.h>
#include <o:\include\conio.h>

#define NCON 2          /* number of simultaneous connections - must be even */
#define BUFMAX 100      /* maximum buffer size */
#define SKT 0x40C       /* the well-known socket we use */
#define NTRC  100       /* the size of the trace buffer */

enum rb_state { openrcv, connect, send, rcv, discon };

struct l4rb rbs[NCON];     /* all the rb's we use */

struct l4rbx {             /* rb extension;  one for each rb */
    enum {connect1, connect2, connect3, connect4, connect5, connect6,
          receive1, receive2, receive3, receive4, receive5, receive6}
        state;  /* state of this rb */
    char buffer [BUFMAX];  /* buffer for this rb */
    }rbxs[NCON];

struct l4rb *active_list;   /* the head of a list of rb's needing attention */

int his_addr;        /* his arcnet address */
int msg_count = 0;  /* outgoing message counter */

char *l4_status_names[] = {  /* l4 rb status */
"uncon", "busy", "done",
"???", "???", "???"};

char *l4_trace_names[] = {   /* l4 trace entries */

"l4connect",    "l4openrcv",     "l4sendmsg",     "l4rcvmsg",      "l4disconn",
"l4abort",      "l4rcvanr",      "l4xmitanr",     "l4connectrcvd","l4disconned",
"l4rcvintr",    "l4discardfmt", "l4discardseq", "l4sysconnect", "l4badwks",
"l4pkttimeout","l4rcvtimeout","l4retry",       "l4unclaimed",  "l4pktused",
"l4pktsent",    "???",           "???",           "???"};

struct {                         /* trace buffer */
        int code;
        struct l4rb *rb;
        } trace_buf[NTRC];

int trace_ptr = 0;   /* next free location in trace buffer */


l4_trace (code, rb)     /* stub trace routine:  just print */
int code;
struct l4rb *rb;
{
/* printf("%x %s\n",rb,l4_trace_names[code]); */
        trace_buf[trace_ptr].code = code;
        trace_buf[trace_ptr].rb = rb;
        if (++trace_ptr >= NTRC) trace_ptr = 0;
    }
```

```
trace_dump()                    /* print the trace table */
{
int i;
        for (i=0; i<trace_ptr && !kbhit(); ++i) {  /* (bad if table wrapped!) */
                printf("%x %s\n",trace_buf[i].rb,
                        l4_trace_names[trace_buf[i].code]);
                }
        }

zero (ptr, size)                /* zero memory */
addr ptr; int size;
{
        while (size-- > 0) *ptr++ = 0;
        }


anr_rtn (rb)    /* Our Asynchronous Notification Routine */
struct l4rb *rb;
{
short int flags;                        /* saved enable/disable state */

    flags = disable();
    rb->link = active_list;   /* add this rb to the active list */
    active_list = rb;
    enable(flags);
        }


tabort(rb)   /* abort the test */
struct l4rb *rb;
{
    printf("Status %d; terminating.\n", rb->status);
    l4_terminate();
        trace_dump();
        exit(99);
    }


panic (msg)                     /* Internal error */
char *msg;
{
        printf("Assertion failed: %s\n",msg);
        l4_terminate();
        trace_dump();
        exit(98);
        }


init()                  /* Initialization */
{
struct l4rb *rb;
struct l4rbx *rbx;
int i;

    active_list = NIL;
    for (i=0; i<NCON; ++i) {
        rb = &rbs[i];
        rbx = &rbxs[i];
            zero (rb, sizeof(struct l4rb));
        zero (rbx, sizeof(struct l4rbx));
        rb->anr = anr_rtn;                /* point rb to anr routine */
        rb->user = (addr) rbx;            /* point rb to rbx */
        rb->id = RBid;
        if (i < NCON/2) {
                        rbx->state = connect1; /* half are connecting */
                                                /* but are quiescent */
```

```
                }
        else {rbx->state = receive1; /* half are receiving */
                anr_rtn(rb);                    /* and are active */
                }
            }
    }


/*  Make up a message to send */

make_msg(rb, code)

struct l4rb *rb;
int code;
{
struct l4rbx *rbx;
int i;

    rbx = (struct l4rbx *) rb->user;
        rb->sndptr = rbx->buffer;
        rb->sndlength = BUFMAX;
    rb->sndtype = code;
        rbx->buffer[0] = msg_count++;  /* packet counter */
    for (i=1; i<BUFMAX; ++i) rbx->buffer[i] = code;
    }



/*  Move an rb to the next state */

process(rb)

struct l4rb *rb;
{
struct l4rbx *rbx;

    rbx = (struct l4rbx *) rb->user;

    switch (rbx->state) {



    /* The state machine for an rb which is the connection initiator */


connect:

        case connect1:
                panic("connect1 state in l4test process!");

    case connect2:          /* send a connect message */

        make_msg(rb,10);
        rb->ph.dstskt = SKT;
        rb->ph.dsthost[2] = his_addr << 8;
        rb->arcnet = TRUE;
        rbx->state = connect3;
        l4_connect(rb);
        break;

    case connect3:          /* receive a response message */

        if (rb->status != l4st_done) tabort(rb);
        rb->rcvptr = rbx->buffer;
        rb->rcvlimit = BUFMAX;
        rbx->state = connect4;
```

```
        l4_rcvmsg(rb);
        break;

    case connect4:      /* generate a final message */

        if (rb->status != l4st_done) tabort(rb);
        make_msg(rb,12);
        rbx->state = connect5;
        l4_sendmsg(rb);
        break;

    case connect5:          /* disconnect */

        if (rb->status != l4st_done) tabort(rb);
        rbx->state = connect6;
        l4_disconn(rb);
        break;

    case connect6:          /* check disconnect, and go connect again */

        if (rb->status != l4st_uncon) tabort(rb);
        goto connect;


    /*  The state machine for an rb which is a connection receiver */

receive:

    case receive1:          /* receive a connection */

                rbx->state = receive2;
        l4_openrcv(rb);
        break;

    case receive2:          /* receive the message */

        if (rb->status != l4st_done) tabort(rb);
        rb->rcvptr = rbx->buffer;
        rb->rcvlimit = BUFMAX;
        rbx->state = receive3;
        l4_rcvmsg(rb);
        break;

    case receive3:         /* generate a response */

        if (rb->status != l4st_done) tabort(rb);
        make_msg(rb,11);
        rbx->state = receive4;
        l4_sendmsg(rb);
        break;

    case receive4:          /* receive the final message */

        if (rb->status != l4st_done) tabort(rb);
        rb->rcvptr = rbx->buffer;
        rb->rcvlimit = BUFMAX;
        rbx->state = receive5;
        l4_rcvmsg(rb);
        break;

    case receive5:          /* disconnect */

        if (rb->status != l4st_done) tabort(rb);
        rbx->state = receive6;
        l4_disconn(rb);
```

```
        break;

    case receive6:          /* check disconnect, and go receive again */

        if (rb->status != l4st_uncon) tabort(rb);
        goto receive;


        } /* switch */
    }


main()
{
int i;
char ch;
struct l4rb *rb;
short int flags;                    /* saved enable/disable state */

    printf("Test pgm for interrupt-driven L4\n");
    if (!l4_init()) {
        printf("Initialization failed\n");
        return;}

        l4_listen (SKT);

    init();     /* initialize rbs */

    printf("Other station's arcnet address: $");
    while (scanf("%x",&his_addr) != 1 );
    printf("address = $%x\n",his_addr);

        printf("Press ESC to abort\n");
        printf("Press C to start connecting\n");
        printf("\n");

    while (TRUE) {  /* wait for interesting things to happen */

                if (kbhit()) {   /* keyboard character */
                        ch = getch();

                        if (ch == 0x1b) {  /* ESC: abort */
                                printf("Aborting...\n");
                                l4_terminate();
                                trace_dump();
                                exit(97);
                                }

                        if (ch == 'C') {  /* C: start connections */
                                for (i=0; i<NCON; ++i) {  /* look for quiescent rbs */
                                        if (rbxs[i].state == connect1) {  /* who want to connect */
                                                rbxs[i].state = connect2;  /* start them */
                                                anr_rtn(&rbs[i]);
                                                }
                                        } /* for */
                                }

                        } /* kbhit */

        if (active_list) {      /* if there is an rb to process */
            flags = disable();
              rb = active_list;     /* remove it from the active list */
              active_list = rb->link;
              rb->link = NIL;
            enable(flags);
            process(rb);            /* and process it */
```

```
    }
  } /* while TRUE */
} /* main */
```