3Com Ethernet Card Incoming Inspection Verification Diagnostic Tool, vers. 31 Oct '88

The program ETestTool performs physical link level packet transmission and reception between multiple cards on a single machine. The current version allows for multiple machines to perform this test while sharing the same network. In this way, the collision detection and retransmission functionality is verified. Errors detected by the Network Interface Controller (NIC) are timestamped and reported to the screen display.

The current version of ETestTool runs as an MPW tool using MPW 2.0 shell or greater. Due to it's UNIX-like structure, the tool may be easily ported to run under A/UX, if desired.

This program only reports the statistics of network activity; there is little interpretation of card "Pass" or "Fail" done by the program. As such, one should be familiar with the NIC documentation in order to interpret many of the subtle problems that may exist on a card. The following errors are timestamped & reported to the Macintosh II screen and can be optionally saved to a disk file:

- Card RAM failure
- 3Com board rework failure
- Slot manager failure
- CRC, Frame alignment, FIFO overrun errors
- FIFO underrun errors
- Missed packets due to buffer overflows
- Packets lost without an error status reported on either the transmitter or receiver NIC

In addition, the following occurrences are tallied:

- Total packets sent
- Number of packet transmissions lost or corrupted
- Average number of retries per packet transmission
- Number of packets received with CRC errors and/or frame alignment errors
- Number of packets missed due to buffer overflow
- Number of FIFO overruns and underruns
- Number of excessive collisions, out of window collisions
- Number of carrier sense lost & collision detect heatbeat failures
- Number of transmission timeouts
- Number of multicast packets received (none are currently sent by the program)

Starting the Verification Test

Copy the file named ETestTool into the folder which contains the MPW 2.0 (or greater) Shell application. After double-clicking the MPW shell icon to start that application, select the New menu item from the File menu and name the window as you would like the ETestTool result file to appear. In this new window type the following, being certain to use the enter key at the end:

```
ETestTool
```

The test will automatically search for valid etherTalk cards, and will echo each card's slot number. After checking RAM size and verifying the RAM, the test for the 3Com rework is run. Failures are noted before running the packet transfer test. After the rework test, packets are sent from card to card, and errors are noted as they

occur. To get the current error accumulatrion status and to pause, hit the mouse button. Press the mouse button again to continue testing. The cursor will spin a quarter turn for every 125 packets sent. To stop testing, simultaneously press the Command (Apple icon) and the period keys. The results may then be saved by using the Save menu Item under the File menu.

ETestTool Options

ETestTool options are entered at the time of starting the test. Below are a few examples to demonstrate the various options. *The slot closest to the power supply is Slot 9.* Options may appear in any order

ETestTool -a -B -C

The above command runs the ETestTool on only slots A, B, & C

ETestTool -i 2000000 -p 60

The above command runs the ETestTool on all EtherTalk cards, sending two million frames to each card before completing. When the -i option is not specified, packets are sent until the operator cancels it by typing Command-period.

The -p (progress) option specifies that the cummulative error status be updated and displayed automatically every 60 minutes, in this case.

ETestTool -v

The -v (verbose) option command runs the ETestTool showing all status information for every packet sent. This option slows down testing and increases memory requirements considerably.

ETestTool -r

The -r option (reverse dest direction) command resolves the ambiguity as to whether an error was generated by the transmitting or receiving EtherTalk card, when more than two cards are under test. For example, if EtherTalk cards reside in slots A, B, C & D, the packets normally proceed from the left to the right (A->B, B->C, C->D, D->A). If packets are corrupted or lost between slots B & C, reversing test direction may move the corrupted or lost packet error to between B & A (B's transmitter is malfunctioning), or to between D & C (C's receiver is malfunctioning).

Notes on 3Com Rework Testing

Currently, boards which have not been reworked will be reported after the RAM tests have run, after starting the verification test. The current test runs between pairs of cards and can only signal that one of the two cards has not been reworked. If all cards under test are said not to be reworked, it is possible that one of those cards may have been reworked, but could not be located because every other card under test has failed. Try each of the cards against a known working card to verify exactly which cards are not reworked.

A page select test has also been added to insure that accesses to undefined portions of the EtherTalk memory map returns a bus error and does not hang the card.

A test has also been added to test for bus errors on back-to-back Nubus accesses while accessing the EtherTalk control registers. A card master (such as the Macintosh Co-Processor Platform or the AST Smart Card) must reside in the system under test which is making frequent use of the NuBus in order for this test to be valid. If the Etesttool reports a bus error while the test is running, the card has failed the test. It will be neccessary to quit the MPW shell application and restart if the bus error occurs, as the stack will have been unbalenced by the bus error.

Notes on Multiple Machine Testing

When 3 or more machines are simultaneously running 'etesttool,' on one connected network, excessive collisions (>16 per frame) begin to occur due to heavy network loading. The timeout and excessive collision messages are tallied by the internal counters (and seen by using the mouse button), but are not displayed as this causes the data file to be needlessly long.

More information on error interpretation can be found in the DP8390/NS32490 Network Interface Controller description, available from National Semiconductor.

```
#include <stdio.h>
#include <signal.h>
#include <Types.h>
#include <OSUtils.h>

pascal long TickCount()
    extern 0xa975;
pascal char Button()
    extern 0xa974;
pascal void ShowCursor()
    extern 0xA853;

short   verbose = 0;                /* 0: Log only fatal errors, 1: Log all information */
short   progress = 0;               /* print progress info every statusTime seconds */
short   polledMode = 0;             /* default is polled tsr */
long    statusTime;                 /* seconds between status reports if progress != 0 */
char    *progName;                  /* Name of this program.                    */

short   autoslots[6] = {0, 0, 0, 0, 0, 0};
short   autoFlag;
short   reversed = 0;
int     numCards = 0;

int     iterations = 0;             /* -i: # of times each card tested,  0 := forever */

main(argc,argv)
int     argc;
char    **argv;
{
    char    *CPUFlag;
    unsigned long   theSecond;
    unsigned long   lastTime = 0;
    short   i;

    setbuf(stdout, (char *) 0);                 /* don't buffer output */
    ParseArgs(argc, argv);

    CPUFlag = (char *) 0x12F;
    if (*CPUFlag < 2)
        ParamDie("This tool incompatible with 68000 or 68010 processors.", " ");

    ZeroTallies();
    AutoTest();                                 /* first call initializes cards */
    printf("Starting packet transfer test…\nHit mouse button for status & pause…\nUse cmd-period to quit…\n");
    do {
        for (i=0; i<numCards; ++i)
            AutoTest();                         /* send one packet per card */

        if (Button()) {
            PrintStats();
            printf("\nNow paused.  Press mouse button to continue…\n");
            while (!Button());  while (Button());
            printf("Continuing…\n");
        }
        if (iterations) {
          if (!(--iterations) ) {               /* -i option */
            PrintStats();
            printf("\nRequested number of iterations complete. Normal exit.\n");
            exit(0);
          }
        }
        if (progress) {                         /* -m option */
            GetDateTime(&theSecond);
            if (!(theSecond%statusTime) && theSecond != lastTime) {
                PrintStats();
                lastTime = theSecond;
            }
        }
    } while (1);
}

ParseArgs(argc, argv)
int     argc;                           /* Count of command arguments.      */
char    *argv[];                        /* Command argument strings.        */
{
    short temp;
    progName = argv[0];     /* parse program name */
    ++argv;
    --argc;

    for ( ; argc > 0; --argc, ++argv ) {
        if ( argv[0][0] == '-' ) {              /* flags */
            switch ( argv[0][1] ) {
                case '9':               temp = 9;       goto common;
                case 'a':   case 'A':   temp = 0xA;     goto common;
                case 'b':   case 'B':   temp = 0xB;     goto common;
                case 'c':   case 'C':   temp = 0xC;     goto common;
                case 'd':   case 'D':   temp = 0xD;     goto common;
                case 'e':   case 'E':   temp = 0xE;
            common:
                    if (!GetSlotMgrInfo(temp) && !autoslots[temp-9]) {
                        ++numCards;
                        autoslots[temp-9] = 1;
                    }
                    break;

                case 'm':               /* Print progress info. */
                    progress = 1;
```

```
                    if ( --argc <= 0 )
                        ParamDie("Missing minute count between progress reports after ", argv[0]);
                    ++argv;
                    statusTime = atoi(argv[0]) * 60;
                    if (statusTime < 1)
                        ParamDie("Minute count must be an integer greater than zero", " ");
                    break;
                case 'i':
                    if ( --argc <= 0 ) {
                        printf("Missing iteration count after ", argv[0]);
                        exit(1);
                    }
                    ++argv;
                    iterations = atoi(argv[0]);
                    if (iterations < 1)
                        ParamDie("Iteration count must be an integer greater than zero", " ");
                    break;
                case 'p':
                    polledMode = 1; /* poll status register instead of using interrupt driver */
                    break;
                case 'r':
                    reversed = 1;   /* reverse test direction */
                    break;
                case 'v':
                    verbose = 1;    /* Log information about every packet sent */
                    break;
                default:
                    printf("# Usage: #%s [-(9-E)] [-i #Packets] [-minutes #minutes] [-reverse]\n", progName);
                    exit(1);
            }   /* switch ( argv[0][1] ) */
        }       /* if ( argv[0][0] == '-' )  */
        else {
            printf("# Usage: #%s [-(9-E)] [-i #NumPackets] [-minutes #minutes] [-reverse]\n", progName);
            exit(1);
        }
    }           /* for ( ; argc > 0; --argc, ++argv ) */
}

ParamDie(desc1, desc2)
char    *desc1;                         /* 1st half of problem description. */
char    *desc2;                         /* 2nd half of problem description. */
{
    fprintf(stderr, "### %s - %s%s\n", progName, desc1, desc2);
    fprintf(stderr, "### %s aborted.\n", progName);
    exit(1);
}

/*
pascal void BusErrDialog(codeLoc, mode, accessLoc)
long    codeLoc;
short   mode;
long    accessLoc;
{
    fprintf(stderr, "A bus error occurred executing code at or before $%lx\n", codeLoc);
    if (mode & 0x40)
            fprintf(stderr, "Program attempted a read to location $%lx\n", accessLoc);
    else    fprintf(stderr, "Program attempted a write to location $%lx\n", accessLoc);
    fprintf(stderr, "\nDiagTool aborted…\n");
    Install0ldV();
    Install24Int();
    exit(3);
}
*/
```

```
/*  printf(" swap_byte of 0x00fa is");
    j = 0x00fa; swap_bytes(&j);
    PrintNum(j);
    printf("Into CopyData (PSTOP - 16), (PSTART+4)*256");
    InitRAM(curSlot);
    CopyData(curSlot, RAMStart(curSlot)+PSTOP*256-16,
        RAMStart(curSlot)+(PSTART+4)*256, 64);
    Query();
*/
/*  Message("Into SendPacket\n");
    SendPacket(mSlot, sSlot, RAMStart(mSlot)+0x10, 512);
    Query();
    sprintf(st, "Slave Buffersize returned %x\n", BufferSize(sSlot) );  Message(st);
    Query();
    Message("Into Slave regs\n");
    AnalyzeRegs(sSlot);
    DumpRegs(sSlot);
    Query();
    Message("Into slave DumpRAM\n");
    DumpRAM(sSlot, Boundry(sSlot)*256, 32);
    DumpRAM(sSlot, (Boundry(sSlot) + 2)*256, 32);
    sprintf(st, "GetPackLen returned %x\n", GetPacketLength(sSlot) );
    Message(st);
    sprintf(st, "GetPackSource returned %x\n", GetPacketSource(sSlot) );
    Message(st);
    Query();
    Message("Into EchoPacket\n");
    EchoPacket(sSlot);
    sprintf(st, "Slave Buffersize returned %x\n", BufferSize(sSlot) );  Message(st);
    DumpRAM(sSlot, 0, 16);
    Query();
    sprintf(st, "Master Buffersize returned %x\n", BufferSize(mSlot) ); Message(st);
    Message("Into Master regs\n");
    DumpRegs(mSlot);
    Message("Into master DumpRAM\n");
    DumpRAM(mSlot, Boundry(mSlot)*256, 32);
    DumpRAM(mSlot, (Boundry(mSlot) + 2)*256, 32);
*/

int
CheckReception(id)              /* move current ptr past last packet received */
short id;
{
    unsigned char link;                 /* link to next page of ring buffer */

    if (BufferSize(id) == 0)    return NO_ERR;

    link = *(RAMStart(id) + Boundry(id) * 256);
    link &= 0x3f;
    --link;                         /* adjustment */
    if (link < PSTART)      link = PSTOP;
    *(RegAddr(id, 3) ) = link;  /* set new boundry */
    return NO_ERR;
}

/*
    Message("Card slot 0 is far left, 5 is far right)");
    Message("\nenter first card slot #(0-5):");
    gets(numbuffer); j = atoi(numbuffer);
    do {
        curSlot = ((int)slotAddrs[j]>>24) & 0x00ff;
        printf("sID was %x", slotTable[numCards].slotID);
        slotTable[numCards++].slotID = curSlot;
        Message("\nenter next card slot #(0-5), or -1 to start test:");
        gets(numbuffer); j = atoi(numbuffer);
    } while (j != -1);
*/
SearchRAM(id)
short id;
{
    short *ptr;
    int i;

    ptr = (short *) RAMStart(id);
    for (i = 0; i < 0x2000; i++) {  /* check two bytes at a time */
        if (*(ptr + i) == 0xabcd) {
            sprintf(st, "found abcd at %x\n", ptr+i);
            Message(st);
        }
    }
}
/*      if (!(packetsSent % 1000) ) {
            printf("%9d",packetsSent);
            printf(backspaces);
        }

        if (EchoPacket(sSlot) ) {
            while (BufferSize(mSlot))   RemoveNextPacket(mSlot);
            while (BufferSize(sSlot))   RemoveNextPacket(sSlot);
            continue;
        }
        CRSMask = 0;    AnalyzeRegs(sSlot);
        CRSMask = 1;    AnalyzeRegs(mSlot);
*/

#define true32b 1
```

```
#define false32b 0

pascal void SwapMMUMode(c)
    char *c;
    extern 0xA05D;

TestROMRead(slot)       /* routine to test 3Com decode fix */
{
    char c, mode, *aPtr;

    return;
    aPtr = (char *) 0xFEFF0000;     /* point to 1st byte in ROM of Caliente card slot E */
    mode = true32b;
    SwapMMUMode(&mode);
/*  c = *aPtr;                      /* 32 bit read from NuBus address */
    mode = false32b;
    SwapMMUMode(&mode);
}


TestRework()        /* routine to verify rework fix for Rev A to Rev C fix */
                    /* assumes slot(0-5) has already been initialized */
{
    short *romPtr, *ramPtr;
    int i, j;
    short sender, receiver;
    short k, data;

    GetNextPair(&sender, &receiver);

    romPtr = (short *) ( (sender+9)*0x100000 + ROM_START+120);  /* point to $4501 in ROM */
    ramPtr = (short *) ( (sender+9)*0x100000 + RAMOffset);      /* point to 1st byte in RAM */

    for (i=0; i<400; ++i)
        *(ramPtr+i) = 0x5555;

    for (i=0; i<100; i++) {         /* unfixed cards fail 11% of the time */
        error = SendPacket(sender, &(slotTable[receiver].ROMID), &myData, 64+i);


    return;
    *(RegAddr(sender, 6) ) = (char) ((800) >> 8);       /* add 14 for header byte length */
    *(RegAddr(sender, 5) ) = (char) ((800) % 256);

    for (i=0; i<100; i++) {         /* unfixed cards fail 11% of the time */
        *(RegAddr(sender, 7) )   = -1;  /* clear ISR masks */
        *(RegAddr(sender, 0xf) ) = 0;       /* clear interrupt mask register */
        *(RegAddr(sender, 0xd) ) = 0x00;    /* transmit config reg: normal operation */
        *(RegAddr(sender, 0) )   = 0x22;    /* start NIC */
        *(RegAddr(sender, 0) )   = 0x26;    /* & transmit */

        j = 9500;
        while (!(*(RegAddr(sender, 7) ) ) && --j )          /* wait for ISR status */
            k = *(romPtr+j);
        if (!j) printf("Timeout on slot %x\n", slot+9);
    }
    printf("j=%d…\n", j);
    for (i=40; i<400; ++i) {
        if (*(ramPtr+i) != 0x5555) {
            printf("card in slot %x not reworked: (%lx) = %x\n",
                    slot+9, (long)(ramPtr+i), *(ramPtr+i));
            break;
        }
    }
}
```

```
code swab
    clr.l   d0
    move.w  2(a6),d0
    ror.w   #8,d0
    move.l  d0,(a6)
    rts
end-code
code fillw ( pat to cnt - )
    move.l  (a6)+,d0
    addq.w  #1,d0
    lsr.w   #1,d0
    subq.w  #1,d0
    move.l  (a6)+,a1
    move.l  (a6)+,d1
@L1
    move.w  d1,(a1)+
    addq.w  #1,d1
    dbra    d0,@L1
    rts
end-code
code movew ( from to cnt - )
    move.l  (a6)+,d0
    addq.w  #1,d0
    lsr.w   #1,d0
    subq.w  #1,d0
    move.l  (a6)+,a1
    move.l  (a6)+,a0
@L1
    move.w  (a0)+,(a1)+
    dbra    d0,@L1
    rts
end-code
code logH
    tst.l   (a6)
    beq     @x
    movea.w #9,a1
    moveq.l #$F,d0
@1  ror.l   #4,d0
    subq.w  #1,a1
    move.l  d0,d1
    and.l   (a6),d1
    beq     @1
    move.l  a1,(a6)
@x  rts
    end-code
```

```
SQLink  equ 0
SQType  equ 4
SQPrio  equ 6
SQAddr  equ 8
SQParm  equ 12

SQSize  equ 16

SIQType equ 6
ISR     equ $E0020

MMU32bit    equ $0CB2           ;(byte) boolean reflecting current 020 machine MMU mode

AppScratch  equ $a80            ;last 4 bytes of Macintosh appl scratch area

_Debugger       OPWORD  $A9FF


    Print Off
    INCLUDE 'Traps.a'
    Print On
    entry MYQUEUE:Data

XmitInt     Proc
;           _Debugger
            move    SR, -(sp)
            move.l  a2, -(sp)
            move.l  #AppScratch, a2
            move.w  #-1, (a2)
            move.b  #$FF, (a1)          ; clear all of the card's interrupts
            moveq   #1, D0              ; indicate interrupt serviced
;           andi.w  #$F8FF,SR
;           ori.w   #$0200,SR           ; interrupt priority must be 2
            move.l  (sp)+, a2
            move    (sp)+, SR
            rts

; Install slot interrupt queue element
; pascal short InstallInt(short slot);  (slot number from 9 to 0xe)

InstallETInt PROC   EXPORT
            move.l  (sp)+,d1            ; pop return address
            clr.l   d0
            move.w  (sp)+,d0            ; pop slot #
            move.l  d1, -(sp)           ; return the return address
            lea     MYQUEUE,a0          ; Ptr to Queue element
            move.l  d0, d1
            mulu    #16, d1             ; point to correct queue element
            adda    d1, a0
            lea     XmitInt,a1          ; Ptr to interrupt routine
            move.l  a1,SQAddr(a0)       ; store interrupt routine in Queue element
            move.w  #100, SQPrio(a0)    ; set interrupt priorty
            move.w  #SIQType, SQType(a0)    ; type of interrupt
            move.l  d0, d1
            lsl.w   #4, d1              ; d1 <- 000000s0
            swap    d1                  ; d1 <- 00s00000
            add.l   #ISR, d1
            move.l  d1, SQParm(a0)      ; Iocation of card's interrupt status reg
            _SIntInstall                ; install interrupt handler
            move.w  d0, 4(sp)           ; return status
            rts


; Remove slot interrupt queue element
; pascal short RemoveInt(short slot);   (slot number from 9 to 0xe)
RemoveETInt Proc Export
            move.l  (sp)+,d1            ; pop return address
            move.w  (sp)+,d0            ; pop slot #
            move.l  d1, -(sp)
            lea     MYQUEUE, a0
            move.l  d0, d1
            mulu    #16, d1             ; point to correct queue element
            adda    d1, a0
            _SIntRemove
            move.w  d0, 4(sp)
            rts

    MACRO
    _m32
    move.l  d0, -(a7)
    move.l  #1, d0
    _SwapMMUMode
    move.l  (a7)+, d0
    ENDM

    MACRO
    _m24
    move.l  d0, -(a7)
    move.l  #0, d0
    _SwapMMUMode
    move.l  (a7)+, d0
    ENDM

; put 68020 into 32 bit addressing mode
Mode32  PROC    EXPORT
    move.l  A1, -(A7)
    movea.l #MMU32bit, a1
    tst.b   (a1)            ;set if 32 bit mode flag set
```

```
    bne     @1              ;don't call swapmmu if already in 32 bit mode
    _m32                    ;switch to 32 bit mode (defined in CommDeclr.h)
@1  move.l  (A7)+, A1
    rts
    ENDP

; put 68020 into 24 bit addressing mode

Mode24  PROC    EXPORT
    move.l  A1, -(A7)
    movea.l #MMU32bit, a1
    tst.b   (a1)            ;set if 32 bit mode flag set
    beq     @1              ;don't call swapmmu if already in 24 bit mode
    _m24
@1  move.l  (A7)+, A1
    rts
    ENDP


;===========================================================================
;   Function GetAddr32(addr32:Ptr; Var Value:Integer): Integer;
;   Author: Bill Weigel
;   passes back -1 if a card resides in given slot, and Value is valid
;               0 if bus error occured accessing ROM space (Value is not valid)
;       Note: This routine assumes that the Bus error replacement routine is in place…

GetAddr32   PROC    EXPORT          ;look for comm card ROM identifier
            IMPORT  BusErrFlag:DATA ;defined in exceptions.a
            move.l  (sp)+, d0       ;pop return address
            move.l  (sp)+, d1       ;pop value ptr
            move.l  (sp)+, a0       ;pop 32 bit addrress
            move.l  d0, -(sp)       ;restore return addr
            lea     BusErrFlag, a1
            clr.w   (a1)            ;if bus error flag == 0, set flag w/o alert
            jsr     mode32          ;prepare for 32 bit address
            move.l  d1, a1
            move.l  #0, d0          ;the following NuBus address instruction must be 4 bytes long
            move.w  (a0, d0.l), (a1) ;store value or cause a bus error
            lea     BusErrFlag, a1
            cmp.w   #-1, (a1)
            beq     @1              ;bus error...
            jsr     mode24          ;fetched a valid address
            move.w  #1, (a1)        ;put up alert if busErr occurs later
@2          move.w  #-1, 4(sp)      ;pass back True
            rts                     ;return
@1          jsr     mode24          ;back to 24 bit addressing
            move.w  #0, 4(sp)       ;pass back False
            rts                     ;return

; Slot queue element parameter block
;MYQUEUE        Record
;           DC.L    0
;           DC.L    0
;           DC.L    0
;           DC.L    0

; Slot queue element parameter block
MYQUEUE     Record
            DS.L    16 * 4      ; space for 16 Slot queue elements
            ENDP

        END
```

```
hd20:Ether_Folder:Ether.c.o ƒ hd20:Ether_Folder:Ether.c
    c hd20:Ether_Folder:Ether.c
hd20:Ether_Folder:MacEther.c.o ƒ hd20:Ether_Folder:MacEther.c
    c hd20:Ether_Folder:MacEther.c
etest ƒ hd20:Ether_Folder:MacEther.c.o hd20:Ether_Folder:Ether.c.o
    Link hd20:Ether_Folder:Ether.c.o hd20:Ether_Folder:MacEther.c.o hd20:MPW_Folder:CLibraries:CRuntime.o hd20:MPW_Folder:CL
    Duplicate -y etest 'ENET TEST:etest'
    eject 'ENET TEST:'
```

```
/*
   This program is the ethernet main driver and menu controller
   */

#include <segload.h>
#include <Types.h>
#include <QuickDraw.h>
#include <Windows.h>
#include <Menus.h>
#include <Events.h>
#include <TextEdit.h>
#include <StdIO.h>
#include <OSUtils.h>
#include <Fonts.h>
#include <Desk.h>
#include <Dialogs.h>

#define applemenu   1
#define testmenu    2
#define nodemenu    3
#define automenu    4
#define slavemenu   5
#define transmitmenu 6
#define receivemenu 7

#define autoitem    1
#define statsitem   2
#define regsitem    3
#define verboseitem 4
#define pauseitem   5
#define reverseitem 6
#define quititem    7

#define machitem    1
#define mach_9slt   2
#define mach_Aslt   3
#define mach_Bslt   4
#define mach_Cslt   5
#define mach_Dslt   6
#define mach_Eslt   7

#define auto_9slt   1
#define auto_Aslt   2
#define auto_Bslt   3
#define auto_Cslt   4
#define auto_Dslt   5
#define auto_Eslt   6
#define clearitem   7

#define slave_9slt  1
#define slave_Aslt  2
#define slave_Bslt  3
#define slave_Cslt  4
#define slave_Dslt  5
#define slave_Eslt  6

/* #define eventMask mDownMask+keyDownMask+autoKeyMask */
#define eventMask -1

/*
 * HIWORD and LOWORD macros, for readability.
 */

#define HIWORD(aLong)       (((aLong) >> 16) & 0xFFFF)
#define LOWORD(aLong)       ((aLong) & 0xFFFF)

#define OFF         0
#define ON          -1
#define FALSE       0
#define TRUE        -1

MenuHandle mymenus[7];
Rect screenrect, dragrect, prect, crect, srect;
short doneflag;
Boolean temp;
EventRecord myevent;
short code, refnum;
short thememu, theitem;
TEHandle hte;
Rect windowrect;
WindowPtr mywindow,whichwindow;
Ptr wrecord;
Point mousepoint;
GrafPort thePort;

short machnum;          /* these parameters passed to ether.c */
short autoslots[6];
short autoFlag;
short verbose = 0;
short reversed = 0;

short curSlot;
short slaveSlot;
short keyhit;
short pauseflag;


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

```
Menu routines                                   */

mastertest(){}

slavetest(){}

restart()
{
    pauseflag = OFF;
}

doquit()
{
   keyhit = TRUE;
   doneflag = ON;
}

setmachnum()
{
    char numbuffer[10];
    printf("\nMachine # = %d\n",machnum);
    printf("Enter machine # (1 - 255): ");
    gets(numbuffer);
    machnum = atoi(numbuffer);
}

setupmenus()
{
    short i;
    char appletitle[2];
    InitMenus();
    appletitle[0] = appleMark;
    appletitle[1] = 0;
    mymenus[0] = (MenuHandle) NewMenu(applemenu, appletitle);
    AddResMenu(mymenus[0],(ResType) 'DRVR');
    mymenus[1] = NewMenu(testmenu,"Test");
    AppendMenu(mymenus[1],"Start;Status;Dump Registers;Verbose;Pause;Reverse Test Direction;Quit");
    mymenus[2] = NewMenu(nodemenu,"Node");
    AppendMenu(mymenus[2],"(Machine;(slot-9;(slot-A;(slot-B;(slot-C;(slot-D;(slot-E");
    mymenus[3] = NewMenu(automenu,"Setup");
    AppendMenu(mymenus[3],"slot-9;slot-A;slot-B;slot-C;slot-D;slot-E;Clear");
    mymenus[4] = NewMenu(slavemenu,"Slave");
    AppendMenu(mymenus[4],"slot-9;slot-A;slot-B;slot-C;slot-D;slot-E");
    mymenus[5] = NewMenu(transmitmenu,"Transmit");
    AppendMenu(mymenus[5],"Send Broadcast;Inhibit CRC;Auto Transmit Disable;Collision Offset Enable");
    mymenus[6] = NewMenu(receivemenu,"Receive");
    AppendMenu(mymenus[6],"Save Errored Packets;Accept Runt Packets;Accept Broadcast;Accept Multicast;Promiscuous Physical");
    for (i=0; i<= 6; i++)
        InsertMenu(mymenus[i], (short) 0);
    DisableItem(mymenus[0], 0);
    DisableItem(mymenus[2], 0);
    DisableItem(mymenus[4], 0);
    DisableItem(mymenus[5], 0);
    DisableItem(mymenus[6], 0);
    DrawMenuBar();
}

docommand(theResult)
long theResult;
{
    char name[256];
    short i;
    char *markChar;
    MenuHandle curMenu;

    theitem = LOWORD(theResult);
    themenu = HIWORD(theResult);
    curMenu = mymenus[themenu -1];

    switch (themenu) {
    case applemenu:
         GetItem(curMenu, theitem, name);
         pauseflag = ON;
         refnum = OpenDeskAcc(name);
         pauseflag = OFF;
         break;
    case testmenu:
         switch(theitem) {
            case autoitem:
                 CheckItem(curMenu, theitem, (Boolean)TRUE);
                 CheckItem(curMenu, pauseitem, (Boolean)FALSE);
                 autoFlag = TRUE;
                 pauseflag = 0;
                 break;
            case statsitem:
                 PrintStats();
                 break;
            case regsitem:
                 dumpregisters();
                 break;
            case verboseitem:
                 verbose = !verbose;
                 CheckItem(curMenu, theitem, (Boolean)verbose);
                 break;
            case pauseitem:
                 pauseflag = !pauseflag;
                 CheckItem(curMenu, theitem, (Boolean)pauseflag);
                 if (!autoFlag && !pauseflag)
```

```
                    printf("\nSelect Auto to begin test\n");
                break;
            case reverseitem:
                reversed = !reversed;
                CheckItem(curMenu, theitem, (Boolean)reversed);
                PrintStats();
                if (reversed) Message("Packets now sent from right to left.\n");
                else          Message("Packets now sent from left to right.\n");
                ZeroTallies();
                break;
            case quititem:
                doquit();
                break;
            }
        break;
    case nodemenu:
        switch(theitem) {
            case machitem:
                setmachnum();
                break;
            default:
                if (curSlot != 0)
                    CheckItem(curMenu,curSlot,(Boolean) FALSE);
                CheckItem(curMenu,theitem,(Boolean) TRUE);
                curSlot = theitem;
                break;
            }
        break;
    case automenu:
        autoFlag = 0;       /* turn off any previous test */
        CheckItem(mymenus[1], autoitem, (Boolean) FALSE);
        switch(theitem) {
            case clearitem:
                for (i = 1; i <= auto_Eslt;i++) {
                    CheckItem(curMenu,i,(Boolean) FALSE);
                    autoslots[i-1] = 0;
                }
                break;
            default:
                autoslots[theitem-1] = !autoslots[theitem-1];
                CheckItem(curMenu,theitem,(Boolean) autoslots[theitem-1]);
                break;
            }
        break;
    case slavemenu:
        if (slaveSlot != 0)
            CheckItem(curMenu,slaveSlot,(Boolean) FALSE);
        CheckItem(curMenu,theitem,(Boolean) TRUE);
        slaveSlot = theitem;
        break;
    }
    HiliteMenu((short) 0);
}

/*  This is the event handling code from main.
*/

PollSystem()
{
    SystemTask();
    temp = GetNextEvent(eventMask, &myevent);
    keyhit = FALSE;
    switch (myevent.what) {
        case mouseDown:
            GetMouse(&mousepoint);
            code = FindWindow(&myevent.where, &whichwindow);
            switch (code) {
                case inMenuBar:
                    docommand(MenuSelect(&myevent.where));
                    break;
/*      to be implemented!!!
                case inSysWindow:
                    SystemClick(&myevent, whichwindow);
                    break;
                case inDrag:
                    DragWindow(whichwindow, &myevent.where, &dragrect);
                    break;
                case inGrow:
                case inContent:
                    break;
*/
                default:
                    break;
                }
        case keyDown:
        case autoKey:
            keyhit = TRUE;
            break;
/*
        case activateEvt:
            if (myevent.modifiers & 1)
              TEActivate(hte);
            else
              TEDeactivate(hte);
            break;
        case updateEvt:
            SetPort(mywindow);
            BeginUpdate(mywindow);
            TEUpdate(&mywindow->portRect, hte);
```

```
            EndUpdate(mywindow);
            break;
*/
        default:
            break;
        }
}

main()
{

    InitGraf(&qd.thePort);
    InitFonts();
    FlushEvents(everyEvent, 0);
    InitWindows();
    TEInit();
    InitDialogs(nil);
    InitCursor();
    setbuf(stdout, (char *) 0);

    SetRect(&screenrect, 0,0, 640, 480);
    SetRect(&dragrect, 4, 24, screenrect.right-4, screenrect.bottom-4);

    doneflag = 0;

    mywindow = (WindowPtr) NewWindow(
                (Ptr) wrecord,
                &screenrect,
                "EtherCard Test",
                (Boolean) TRUE, (short) noGrowDocProc, (WindowPtr) -1,
                (Boolean) TRUE, (long) 0);
    SetPort(mywindow);

    setupmenus();

    prect = mywindow->portRect;
    prect.top = mywindow->portRect.top;
    prect.right = mywindow->portRect.right;
    prect.left = mywindow->portRect.left;
    prect.bottom = mywindow->portRect.bottom;
    InsetRect(&prect, 4, 0);
    hte = (TEHandle) TENew(&prect, &prect);

    pauseflag = OFF;
    slaveSlot = 0;
    curSlot = 0;
    machnum = 0;

    /* following two lines added to support autostart */
/*   autoFlag = TRUE;
     pauseflag = 0;
*/


    do {
        if (!pauseflag) {
            if (autoFlag)   AutoTest();
        }
        PollSystem();
    } while (doneflag == 0);

    CloseWindow(mywindow);
    QuitTest();
    Egress();
}

Egress() {              /* generate MacsBugs symbol */
    ExitToShell();
}
```

```
/*  Ethercard verification routines and diagnostics in MPW C
    Copyright © 1987, 1988 Apple Computer, Inc.
    All rights reserved

    Mod History:
    20 May 87   BW  1st Rev.
    28 May 87   BW  Added Mac interface
    4 June 87   BW  Added ROMID field facilitating multiple test machines on
                    a single network (to test collision detection & retransmission)
    10 June 87  BW  Added verbose, reverse test direction commands.
    18 April 88 BW  Added code to support ethernet packet analyzer.
    4  May   88 BW  Single card test scenario to locate cause of missing packet problem.
    10 June  88 BW  Added test for 'upgrade' rework for incoming inspection.
    20 June  88 BW  Added FindCard routine.
    28 June  88 BW  Mods to support running as a MPW tool
    8 August 88 BW  Added Slot Mgr testing
    26 Sept  88 BW  Fixed TestROM call, RAM error location print statement
    21 Nov   88 BW  Integrated interrupt / polled transmit option, also page select and
                    BlastTCR tests

*/

#include <stdIO.h>
#include <Types.h>
#include <Events.h>
#include <Files.h>
#include <OSUtils.h>
#include <Slots.h>
#include <errors.h>



#define min(A, B)    ((A) < (B)) ? (A) : (B)
#define TIME_ZIPS_BY -1
#define MAX_SLOTS 6
#define REENTER 0x1939

char copyright[] = "Copyright © 1987, 1988 Apple Computer, Inc.";

/*
pascal void Debugger()
    extern 0xa9ff;
*/

pascal short InstallETInt(slot)
    short slot;
    extern;

pascal short RemoveETInt(slot)
    short slot;
    extern;

pascal void SystemTask()
    extern 0xa9b4;

pascal void RotateCursor(tick)          /* Rotates beach ball cursor.       */
   long tick;
   extern;
pascal void SpinCursor(tick)            /* Rotates beach ball cursor. uses internal counter. */
   short tick;
   extern;

# define    CURSORFORWARD       (32)    /* Tell RotateCursor to go forward. */
# define    CURSORBACKWARD      (-32)   /* Tell RotateCursor to go backward.*/


pascal  void    InstallMyErrV()         extern;         /* defined in exceptions.a */
pascal  void    InstallOldV()           extern;         /* defined in exceptions.a */

typedef struct {
    short l;            /* uses the low order byte of the unique ROM id */
    short m;
    short h;
} romID;
romID broadcastID = {
    0xffff,
    0xffff,
    0xffff
};
unsigned short triggerData = 0xABCD;        /* sent in a packet after error detected */
unsigned short regularData = 0x0000;        /* sent in a packet after no error detected */

enum statvals {SENT_PACK, RECV_PACK, TIMEOUT, WAITING};

struct slotinfo {
    romID ROMID;
    short slotID;           /* slot location (0-5, presently same as index) */
    char master;            /* master card if non-zero */
    enum statvals status;
    unsigned int lastReceptionNum;  /* Packet number of last valid reception */
    int timeout;            /* currently a loop decrementing counter */
}   slotTable[MAX_SLOTS];

#define TO_SCREEN 0x01
#define TO_FILE 0x02
#define TO_PRINT 0x04
short outputDirection = TO_SCREEN;
FILE *outFile;              /* log errors to file */
char *fileName = "error.out";
```

```
void    MyAlert(),
        Query(),
        DumpRAM(),
        DumpRegs();

char    *RegAddr(),         /* these functions return ethernet registers */
        *RAMStart(),        /* and memory addresses */
        *memcpyl(),
        *GetSlotAddress();

char    *TimeStamp();

int ignoreErr;              /* temporarily disable error reporting mechanism */

unsigned int packetsSent;   /* frames sent successfully */
unsigned int Retries;       /* used to calculate Retries/Packet */
unsigned int packetsReceived;
unsigned int multicastsReceived[MAX_SLOTS];
unsigned int heartFailures[MAX_SLOTS];
unsigned int carrierLost[MAX_SLOTS];
unsigned int collisions[MAX_SLOTS];
unsigned int OWC[MAX_SLOTS];
unsigned int CRCErrorTally[MAX_SLOTS];
unsigned int CRCErrors[MAX_SLOTS];
unsigned int FIFOunderruns[MAX_SLOTS];
unsigned int FIFOoverruns[MAX_SLOTS];
unsigned int receiveErrors[MAX_SLOTS];
unsigned int alignErrors[MAX_SLOTS];
unsigned int excessCollisions[MAX_SLOTS];
unsigned int lengthErrors[MAX_SLOTS];
unsigned int missedPackets[MAX_SLOTS];
unsigned int bufferOverflows[MAX_SLOTS]; /* Packets lost due to lack of resources */
unsigned int transmitTimeouts[MAX_SLOTS];
unsigned int lostOrCorrupt[MAX_SLOTS];

char *errorAddress;         /* set by routine detecting the problem */
char st[256];
int CRSMask;                /* mask carrier sense lost bit on receiving end */

extern short polledMode;    /* if non-zero, use polled status reg for transmit complete */

extern int numCards;        /* defined in EtherTool.c */
extern short machnum;       /* machine number */
extern short autoFlag;
extern short autoslots[];
extern short verbose;       /* when 0, only fatal errors are logged */
extern short reversed;      /* reverse transmit card to resolve error ambiguity */


extern struct data {
    short BusErrFlag;
    short BusErrRetry;
} data;


short sSlot;        /* valid range is presently 0 to 5 */
short mSlot;        /* master (sender) slot */
char *slotAddrs[6] =        /* address of 1st byte in each slot */
    {   (char *)0xf9900000, (char *)0xfaa00000, (char *)0xfbb00000,
        (char *)0xfcc00000, (char *)0xfdd00000, (char *)0xfee00000 };


dumpregisters()
{
    int i;

    if (!numCards) {
        SysBeep(3);
        return;
    }
    SetTest();
    for (i=0; i < MAX_SLOTS; i++)
        if (autoslots[i])
            DumpRegs(slotTable[i].slotID);
}

SetTest()   /* set proper ids of cards to test */
{
    int i;

    for (i=0; i < 6; i++) {
        if (autoslots[i]) {
            slotTable[i].slotID = i;    /* this relationship may change */
        }
    }
}

void myexit()                       /* cleanup after cmd-. termination */
{
    int i;
/*  printf("Terminating etesttool…\n"); */
    if (!polledMode)
        for (i = 0; i < MAX_SLOTS; i++)
            if (autoslots[i])
                RemoveETInt(i+9);
/*  PrintStats(); */
}
```

```
AutoTest()
{
    int i, oldVerbose, error;
    int seed = 512;
/*  romID theID; */

    atexit(myexit);                 /* used to remove slot interrupts after an abort */
    if (autoFlag != REENTER) {      /* new test run */
        Message("\nEtherTalk Incoming Inspection Diagnostic Tool:");
        Message(" Vers 1.0d8: 23 November 1988\n\n");

        if (!autoslots[0] && !autoslots[1] && !autoslots[2] && !autoslots[3] && !autoslots[4] && !autoslots[5])
            FindCards();
        SetTest();                          /* set proper ids of cards to test */

        printf("\n");
        for (i = 0; i < MAX_SLOTS; i++) {
            if (autoslots[i]) {
                InitSlot(slotTable[i].slotID);  /* initializes NIC, performs RAM testing */
            }
        }

        if (!polledMode)
            for (i = 0; i < MAX_SLOTS; i++)
                if (autoslots[i])
                    if ((error = InstallETInt(i+9) ) )
                        printf("ERROR:Install receive interrupt on slot %X returned %d\n", i+9, error);

        printf("\n");
        for (i = 0; i < MAX_SLOTS; i++)
            if (autoslots[i])
                DataHoldTest(i);

        printf("\nNuBus Bus Error Test (Valid only if MR-DOS running on a separate CPU card)\n");
        for (i = 0; i < MAX_SLOTS; i++) {
            if (autoslots[i]) {
                printf("Testing Slot %X…\n", i+9);
                BlastTCR(i);
            }
        }

        PageSelectTest();
        if (numCards < 2) {
            Message("2 or more connected 3Com/Apple EtherTalk cards needed to continue testing…\n");
            autoFlag = 0;
            exit(1);
            return;
        }
        SlotDecodeRework();
        ZeroTallies();                  /* reset packet counter from rework test */
        autoFlag = REENTER;
    }       /* if (autoFlag != REENTER) */
    do {

        if (reversed)   GetNextPair(&sSlot, &mSlot); /* resolve transmit -    */
        else            GetNextPair(&mSlot, &sSlot); /* receive err ambiguity */
        if (++seed > 1500) {
            seed = 64;
        }
        if (!(packetsSent % 125) ) {
            RotateCursor(CURSORFORWARD);
        }

        error = SendPacket(mSlot, &(slotTable[sSlot].ROMID), &regularData, seed);

        if (error) {    /* timeout or transmit abort error */
            oldVerbose = verbose;
            verbose = -1;
            AnalyzeRegs(mSlot);
            verbose = oldVerbose;
            while (BufferSize(mSlot))   RemoveNextPacket(mSlot);
            while (BufferSize(sSlot))   RemoveNextPacket(sSlot);
            InitNIC(slotTable[mSlot].slotID);           /* hanging timeout fix */
            InitNIC(slotTable[sSlot].slotID);           /* hanging timeout fix */
            continue;
        }

        if ((!BufferSize(sSlot) /* || GetPacketLength(sSlot) != seed */) &&
             !(*(RegAddr(mSlot, 4) ) & 0x08 ) ) {                   /* not TSR abort */
/*          if (!SendPacket(sSlot, &broadcastID, &triggerData, 64) )    /* trigger analyser */
/*              --packetsSent;          /* don't count trigger packet in totals */
            ++lostOrCorrupt[mSlot];
            Message(TimeStamp() );
            sprintf(st, "\nPacket #%d from %X to %X was corrupted or lost ",
                        packetsSent-1, (int)mSlot+9, (int)sSlot+9);
            Message(st);
            if (!BufferSize(sSlot) )                /* changed m to s 15 April 1988 */
                Message("(receive buffer empty)");
            else if (GetPacketLength(sSlot) != seed) /* changed m to s 15 April 1988 */
                Message("(data length field incorrect)");
            oldVerbose = verbose;
            verbose = -1;
            Message("\n----Transmitter packet header & data----\n");
            DumpRAM(mSlot, (short *)RAMStart(mSlot), 24);
            Message("----Transmitter Slot Status----\n");
            AnalyzeRegs(mSlot);
            Message("----Receiver Slot Status----\n");
            AnalyzeRegs(sSlot);
```

```
            verbose = oldVerbose;
            Message("\n\n");
/*          for (i = 0; i < MAX_SLOTS; i++)
                if (autoslots[i])
                    InitNIC(slotTable[i].slotID); /* hanging card fix */
        }
        CRSMask = 0;    AnalyzeRegs(mSlot); /* also updates error tally count */
        CRSMask = 1;    AnalyzeRegs(sSlot);
        while (BufferSize(mSlot))   RemoveNextPacket(mSlot);
        while (BufferSize(sSlot))   RemoveNextPacket(sSlot);

        ignoreErr = 0;  /* reset disable error reporting mechanism */
        return;         /* return to user interface routine */

    } while (TIME_ZIPS_BY);
}

#define TPSR    0x00                    /* Transmit Page Start */
#define PSTART  (unsigned char)0x08     /* receive buffer start */
/* #define PSTOP    (unsigned char)0x3f /* 1/2 of 64 K RAM      */
#define PSTOP   (unsigned char)0x1f

char *
GetSlotAddress(id)  /* returns slot address of id */
{
    return slotAddrs[id];
}

GetNextPair(mSlot, sSlot)
short *mSlot, *sSlot;
{
    static int lastMaster = -1;

    if (lastMaster < 0)
        while (!autoslots[++lastMaster] )
            ;
    *sSlot = slotTable[lastMaster].slotID;
    slotTable[lastMaster].master = 0;
    do {
        ++lastMaster;
        lastMaster %= MAX_SLOTS;
    } while (!autoslots[lastMaster] );

    *mSlot = slotTable[lastMaster].slotID;
    slotTable[lastMaster].master = -1;
}

int
SendPacket(sID, dROMID, data, dLength)          /* returns non-zero on failure */
short sID;
romID *dROMID;
short *data;     /* In this version, only 2 bytes copied to xmit buffer so as to maximize traffic */
short dLength;
{
    short *wPtr;

    wPtr = (short *) ((long)RAMStart(sID) & 0xfffffff0);
    *wPtr =       dROMID->h & 0x00ff;   /* mask extra bits in this version */
    *(wPtr + 1) = dROMID->m & 0xff00;
    *(wPtr + 2) = dROMID->l & 0xff00;   /* destination address */

    *(wPtr + 3) = slotTable[sID].ROMID.h & 0x00ff;
    *(wPtr + 4) = slotTable[sID].ROMID.m & 0xff00;
    *(wPtr + 5) = slotTable[sID].ROMID.l & 0xff00;  /* source address */
    *(wPtr + 6) = dLength;
    *(wPtr + 7) = (short)(packetsSent>>16);                 /* packet # data */
    *(wPtr + 8) = (short)(packetsSent & 0xffff);            /* packet # data */
    *(wPtr + 13) = *data;                                   /* test data */
    *(RegAddr(sID, 6) ) = (char) ((dLength+14) >> 8);       /* add header byte length */
    *(RegAddr(sID, 5) ) = (char) ((dLength+14) % 256);

    *(RegAddr(sID, 7) )   = -1;     /* clear receiver ISR masks 21 nov 88 */

    return Transmit(sID);
}

int
Transmit(id)
{
    int i;
    unsigned int tCount;
    short   *xmitComplete;

    xmitComplete = (short *) 0xa80; /* last 4 bytes of AppScratch area */
    *xmitComplete = 0;              /* set non-zero by xmit complete interrupt */

    tCount = (unsigned int) TickCount();    /* 8 second timeout was requested by 3Com engineers */
    *(RegAddr(id, 7) )   = -1;      /* clear ISR masks */
    if (polledMode)
        *(RegAddr(id, 0xf) ) = 0;   /* clear interrupt mask register */
    else
        *(RegAddr(id, 0xf) ) = 0x0A; /* interrupt mask set for transmit & xmit error */
    *(RegAddr(id, 0xd) ) = 0x00;    /* transmit config reg: normal operation */
    *(RegAddr(id, 0) )   = 0x22;    /* start NIC */
    *(RegAddr(id, 0) )   = 0x26;    /* & transmit */

/*  TestROMRead(id);                /* test for decode address problem */
/*  for (i=0; i<100; ++i)       ;   /* kludge wait before testing status (was 2000) */
```

```
    if (polledMode) {
        if (*(RegAddr(id, 7) ) ) {      /* if any bits set, wait a bit longer */
            for (i=0; i<2000; ++i)  ;   /* kludge wait before testing status (was 2000) */
        }
        while (!(*(RegAddr(id, 7) ) ) && (unsigned int)TickCount() - tCount < 8*60 )
            TestROMRead(id);            /* wait for ISR status */
    }
    else {                          /* transmit interrupt mode */
        while (!(*xmitComplete) && (unsigned int)TickCount() - tCount < 8*60 )
            TestROMRead();              /* wait for ISR status */
    }
    if ((unsigned int)TickCount() - tCount >= 8*60) {
        Message(TimeStamp() );
        sprintf(st, "\nSlot %X, Packet #%d: NIC transmit timeout\n", id+9, packetsSent-1);
        Message(st);
        ++transmitTimeouts[id];
/*      for (i = 0; i < MAX_SLOTS; i++)
            if (autoslots[i])
                InitNIC(slotTable[i].slotID);           /* hanging timeout fix */
        return -1;
    }
    else if ( ( *(RegAddr(id, 7)) & 0x08) ) {           /* transmit aborted bit */
        printf("Slot %X, packet %d: Transmit aborted bit set\n", id+9, packetsSent-1);
        return 0;
    }
    else
        ++packetsSent;
    return 0;
}

int
EchoPacket(id)      /* send a packet back to its source */
short id;           /* returns non-zero on error */
{
    int cnt;
    romID theROMID;

    if (!BufferSize(id) ) {
        printf("EchoPacket called with no packets in buffer\n");
        return -1;
    }

    cnt = GetPacketLength(id);
    CopyData(id, RAMStart(id) + Boundry(id)*256 + 4, RAMStart(id), cnt);
    GetPacketSource(id, &theROMID);
    if (SendPacket(id, &theROMID, RAMStart(id)+16, cnt) )
        return -1;                  /* handle this error in main loop */
    RemoveNextPacket(id);           /* clean up buffer */
    return 0;
}

unsigned char
GetNextLink(id)         /* returns the next packet's page address */
{
    return *((unsigned char *)RAMStart(id) + Boundry(id)*256 + 1) & PSTOP;
}

RemoveNextPacket(id)    /* clear next packet from ring buffer without saving */
short id;
{
    unsigned char link;

    if (!BufferSize(id) ) {
/*      Message("RemoveNextPacket called with no packets in buffer\n"); */
        return -1;
    }
    link = GetNextLink(id);
    if (--link < PSTART) link = PSTOP; /* wraparound adjust */
/*  printf("removeNP slot %X link is %x\n", id+9, (int)link);   */
    ++packetsReceived;  /* should this be elsewhere??? */
    *(RegAddr(id, 3) ) = link;  /* reset NIC boundry pointer */
}

GetPacketLength(id)
{
    short *cnt;

    cnt = (short *)(RAMStart(id) + Boundry(id)*256 + 16);   /* from receive buffer */
    return *cnt;
}

GetPacketSource(id, source)
short id;
romID *source;      /* returns the sender ROM id of next packet in memory */
{

    if (BufferSize(id) == 0) {      /* maybe call an alert??? */
        Message("Uh oh, GPS called with no packets in buffer...\n");
        return -1;
    }
    source->l = *(short *)(RAMStart(id) + Boundry(id)*256 + 0xe);
    source->m = *(short *)(RAMStart(id) + Boundry(id)*256 + 0xc);
    source->h = *(short *)(RAMStart(id) + Boundry(id)*256 + 0xa);
    return 0;
}

int
Boundry(id) /* returns page pointer to next package to remove from ring */
short id;
```

```
{
    unsigned char bPtr;

    *(RegAddr(id, 0) ) = 0x22;                      /* page 0 */
    bPtr = *((unsigned char *)RegAddr(id, 3) );
    if (++bPtr > PSTOP) /* ++bPtr points to next available buffer */
        bPtr = PSTART;
    return (int)bPtr;
}

int
Current(id) /* returns current page register (head ptr to receive ring buffer) */
short id;
{
    char curPtr;

    *(RegAddr(id, 0) ) = 0x62;                      /* page 1 */
    curPtr = *(RegAddr(id, 7) );
    *(RegAddr(id, 0) ) = 0x22;
    return (int)curPtr;
}

int
BufferSize(id)  /* returns the page size of unread buffers */
short id;
{
    int num;

    num = Current(id) - Boundry(id);
    if (num < 0) num += PSTOP - PSTART;
    return num;
}

struct NICpair { /* each NICpair pair corresponds to data, register */
    char data;
    char reg;
} NICinit[] = {
    0x22, 0x0,  /* NIC off-line */
    0x21, 0x0,  /* init command register, abort DMA, NIC off-line, Page 0 */
    0x49, 0xE,  /* Data config set - word length DMA transfers */
    0x00, 0xA,  /*clear RBCR0 */
    0x00, 0xB,  /*clear RBCR1 */
    0x04, 0xC,  /* Rcv config (accept broadcast) */
    0x02, 0xD,  /* NIC in loopback mode */
    PSTART,0x1, /* set PSTART reg - ring buffer init */
    PSTOP, 0x2, /* set PSTOP reg - ring buffer init */
    PSTOP, 0x3, /* set boundary reg - ring buffer init */
    TPSR, 0x4,  /* Xmit Page Start */
    0xFF, 0x7,  /* reset Interrupt Status register */
    0x00, 0xF,  /* clear Interrupt Mask register */
/*  0x02, 0x6,  /* set Xmit byte count register 1 */
/*  0x00, 0x5,  /* set Xmit byte count register 0 */
    0x61, 0x0,  /* select page 1 registers */
    0x00, 0x1,  /* set Physical Address Regs to node addr - regs 1 -> 6 */
    0x00, 0x2, 0x00, 0x3, 0x00, 0x4, 0x00, 0x5, 0xFF, 0x6,
    0x00, 0x8,  /* set Multicast Address regs - regs 8 -> F */
    0x00, 0x9, 0x00, 0xA, 0x00, 0xB,
    0x00, 0xC, 0x00, 0xD, 0x00, 0xE, 0x00, 0xF,
    PSTART, 0x7,/* curr page ptr to ring buffer */
    0x21, 0x0,  /* Select Page 0 regs */
    0x22, 0x0,  /* set start mode */
    0x00, 0xD   /* initialize Xmit config register */
};

#define NO_ERR  0x00
#define RAM_ERR 0x01
#define OUT_OF_BOUNDS 0x02
#define ISR_RXE 0x03
#define ISR_TXE 0x04
#define ISR_OVW 0x05
#define ISR_CNT 0x06
#define TSR_COL 0x07
#define TSR_ABT 0x08
#define TSR_CRS 0x09
#define TSR_FU  0x0a
#define TSR_CDH 0x0b
#define TSR_OWC 0x0c
#define RSR_CRC 0x0d
#define RSR_FAE 0x0e
#define RSR_FO  0x0f
#define RSR_MPA 0x10
#define RSR_PHY 0x11
#define RSR_MUL 0x12
#define RSR_PRX 0x13
#define TSR_PTX 0x14
#define TSR_DFR 0x16
#define ISR_PTX 0x15
#define ISR_PRX 0x17
#define ISR_RDC 0x18

#define FINFO_ERR 0x25

#define NICOffset (char *)0xe003c

char *
RegAddr(id, n)      /* returns address of NIC register n in slot id */
short id, n;
{
    return slotAddrs[id] + NICOffset - n*4;
```

```
}

InitSlot(id)
short id;
{
    short i;

    *(RegAddr(id, 0) ) = 0x21;      /* turn off NIC during RAM test */
    MyAlert(id, TestRAM(id) );
    InitNIC(id);
    for (i=0; i < 8; i++)           /* the first 16 bytes */
        *((int *)RAMStart(id) + i) = 0; /* zero xmit buffer */
}

int
InitNIC(id)
short id;
{
    int i;

    for (i=0; i < sizeof(NICinit)/sizeof(struct NICpair); i++)
        *(RegAddr(id, NICinit[i].reg) ) = NICinit[i].data;

    *(RegAddr(id, 0) ) = 0x61;          /* Page 1 registers */
    slotTable[id].ROMID.h = *(short *)(GetSlotAddress(id) + 0xf0006);
    slotTable[id].ROMID.m = *(short *)(GetSlotAddress(id) + 0xf0008);
    slotTable[id].ROMID.l = *(short *)(GetSlotAddress(id) + 0xf000a);
    if (slotTable[id].ROMID.h == 0x700 && slotTable[id].ROMID.m == 0x900 &&
        slotTable[id].ROMID.l == 0xb00) {
        sprintf(st, "WARNING: card in slot %X may be missing ROM identifier chip.\n", id+9);
        Message(st);
        Message("Program cannot run without this ROM installed.\n");
        Query();
    }
    *((int *)RegAddr(id, 1) ) = 0;      /* multicast address has MSB set high */
    *((int *)RegAddr(id, 2) ) = ((int)slotTable[id].ROMID.h) << 16;
    *((int *)RegAddr(id, 3) ) = ((int)slotTable[id].ROMID.m) << 16;
    *((int *)RegAddr(id, 4) ) = 0;
    *((int *)RegAddr(id, 5) ) = ((int)slotTable[id].ROMID.l) << 16;
    *((int *)RegAddr(id, 6) ) = 0;

    slotTable[id].ROMID.h = (slotTable[id].ROMID.h >> 8) & 0x00ff;
                    /* so as not to be confused with a multicast address */
/*  slotTable[id].ROMID.m &= 0xff00;
    slotTable[id].ROMID.l &= 0xff00;
*/
                                        /* Physical Addr node # */
    *(RegAddr(id, 0) ) = 0x21;          /* Page 0 regs */
    i = *RegAddr(id, 0x0d);             /* clear tally counters */
    i = *RegAddr(id, 0x0e);
    i = *RegAddr(id, 0x0f);
/*  printf("block id = %x %x %x", *(short *)(GetSlotAddress(id) + 0xf0000),
                            *(short *)(GetSlotAddress(id) + 0xf0002),
                            *(short *)(GetSlotAddress(id) + 0xf0004) );
/*
    printf(" slot id = %x %x %x", *(short *)(GetSlotAddress(id) + 0xf0006),
                            *(short *)(GetSlotAddress(id) + 0xf0008),
                            *(short *)(GetSlotAddress(id) + 0xf000a) );
*/
    *(RegAddr(id, 0) ) = 0x22;          /* start NIC */

    return NO_ERR;
}

#define RAMOffset (char *) 0xD0000

char *
RAMStart(id)    /* return the first location of RAM on board in slot id (0-5) */
short id;
{
    return GetSlotAddress(id) + RAMOffset;
}

int
TestRAM(id) /* returns non-zero error on failure */
short id;
{
    register short *ptr;
    register unsigned short i;
    unsigned short RAMSize;

    ptr = (short *) RAMStart(id);
    *(ptr+0x2000) = 0x1234;             /* see if memory wraps around… */
    *ptr = 0xabcd;
    if ((*(ptr+0x2000) & 0xffff) == 0xabcd)      RAMSize = 0x2000;  /* 16K checked two bytes at a time */
    else if ((*(ptr+0x2000) & 0xffff) == 0x1234) RAMSize = 0x8000;  /* 64K checked two bytes at a time */
    else {
        RAMSize = 0x2000;               /* some cards don't wrap around */
/*      printf("(16K RAM didn't wraparound) "); */
/*      printf("*(ptr+0x2000) = %X ", (*(ptr+0x2000) & 0xffff)); */
    }
    printf("Slot %X Apple/3Com EtherTalk card: %ldK bytes RAM:", id+9, (RAMSize/0x200));
    for (i = 0; i < RAMSize; ++i)
        *(ptr+i) = 0xffff;
    for (i = 0; i < RAMSize; ++i)
        if (*(ptr+i) & 0xffff != 0xffff) {  /* compiler returns 32 bit data */
            errorAddress = (char *) (ptr + i);
            Message("Test 1,");
            return RAM_ERR;
```

```
        }

    for (i = 0; i < RAMSize; ++i)
        *(ptr+i) = 0;
    for (i = 0; i < RAMSize; ++i)
        if (*(ptr+i) != 0) {
            errorAddress = (char *) (ptr + i);
            Message("Test 2,");
            return RAM_ERR;
        }

    for (i = 0; i < RAMSize; ++i) {
        *(ptr+i) = i;
        if (*(ptr+i) != i) {
            errorAddress = (char *) (ptr + i);
            Message("Test 3,");
            return RAM_ERR;
        }
    }

    for (i = 0; i < RAMSize; ++i)
        *(ptr+i) = 0x5555;
    for (i = 0; i < RAMSize; ++i)
        if (*(ptr+i) != 0x5555) {
            errorAddress = (char *) (ptr + i);
            Message("Test 4,");
            return RAM_ERR;
        }

    for (i = 0; i < RAMSize; ++i)
        *(ptr+i) = 0xaaaa;
    for (i = 0; i < RAMSize; ++i)
        if (*(ptr+i) & 0xffff != 0xaaaa) {
            errorAddress = (char *) (ptr + i);
            Message("Test 5,");
            return RAM_ERR;
        }

    printf(" verified.\n");
    return NO_ERR;
}

void
DumpRegs(id)        /* of NIC in slot id (9 - e) */
short id;
{
    int i;
    char j;

    *(RegAddr(id, 0) ) = 0x21;
    sprintf(st,"NIC Register dump of machine %x, slot %X\nPage 0  ", machnum, id + 9);  Message(st);
    for (i=0; i < 16; i++) {
        if (i == 6)
            Message("6:XX ");   /* don't disturb data in FIFO register */
        else {
            j = *(RegAddr(id, i) );
            sprintf(st,"%1x:%02x ", i, (j & 0xff) );    Message(st);
        }
    }
    Message("\nPage 1  ");
    *(RegAddr(id, 0) ) = 0x61;
    for (i=0; i < 16; i++) {
        j = *(RegAddr(id, i) );
        sprintf(st, "%1x:%02x ", i, (j & 0xff) );   Message(st);
    }
    Message("\n");
    *(RegAddr(id, 0) ) = 0x22;              /* page 0 */
}

void
DumpRAM(id, begin, cnt) /* display a range of memory in card id */
short id;
char *begin;
int cnt;
{
    begin = (RAMStart(id) + ((short)begin & 0x3ffe) ); /* start on word boundry */
    sprintf(st,"%4x: ", begin); Message(st);
    while (cnt-- > 0) {
        sprintf(st, "%04x ", *(short *)begin & 0xffff); Message(st);
        begin += 2;
        if (!((short)begin % 16) && cnt) {
            sprintf(st, "\n%4x: ", begin);
            Message(st);
        }
    }
    Message("\n");
    return;
}

char *memcpy1(to, from, cnt)
short *to, *from;
int cnt;
{
    cnt >>= 1;
    while (cnt--)
        *(to++) = *(from++);
    return (char *)from;
}
```

```
CopyData(id, from, to, cnt)
short id;
char *to, *from;    /* absolute addresses */
int cnt;            /* count in bytes */
{
    int first;      /* used to calculate page memory wraparound */

    first = min(RAMStart(id) + (PSTOP+1)*256 - from, cnt);
    memcpy1(to, from, first);
    to += first;
    if ( (cnt -= first) > 0)    memcpy1(to, RAMStart(id) + PSTART*256, cnt);
}

fillw(seed, to, cnt)    /* simple routine for filling a data buffer */
register int seed;
register short *to;
register int cnt;
{
    ++cnt;
    cnt >>= 1;          /* # of words in packet */
    while (cnt--)
        *(to++) = seed++;
    return;
}

void
swap_bytes(i)       /* ror.w #8, d0 */
register short *i;
{
    register short j;

    j = *i & 0xffff;
    *i = j<<8 | j>>8;
    return;
}

AnalyzeRegs(id)     /* check status registers of NIC for errors and such */
short id;
{
    register char flags;

    if (packetsSent < 6)
        return;         /* NIC may show false errors after power on */
/*  *(RegAddr(id, 2) ) = 0x21;  /* stop NIC while checking regs */
    flags = *(RegAddr(id, 7) );
/*  if (flags != 3 && flags != 2)
        printf("ISR = %X, RSR = %X\n", (int) flags, (int) *(RegAddr(id, 0xc) ) );
*/
    if (flags & 0x01)   MyAlert(id, ISR_PRX);
    if (!(flags & 0x02))    MyAlert(id, ISR_PTX);
    if (flags & 0x04)   MyAlert(id, ISR_RXE);
    if (flags & 0x08)   MyAlert(id, ISR_TXE);
    if (flags & 0x10)   MyAlert(id, ISR_OVW);
    if (flags & 0x20)   MyAlert(id, ISR_CNT);
    if (flags & 0x40)   MyAlert(id, ISR_RDC);

    flags = *(RegAddr(id, 4) );
    if (!(flags & 0x01))    MyAlert(id, TSR_PTX);
    if (flags & 0x02)   MyAlert(id, TSR_DFR);
    if (flags & 0x04) { MyAlert(id, TSR_COL); Retries += *(RegAddr(id, 5) ); }
    if (flags & 0x08)   MyAlert(id, TSR_ABT);
    if ((flags & 0x10) && !CRSMask) MyAlert(id, TSR_CRS);
    if (flags & 0x20)   MyAlert(id, TSR_FU);
    if (flags & 0x40)   MyAlert(id, TSR_CDH);
    if (flags & 0x80)   MyAlert(id, TSR_OWC);

    flags = *(RegAddr(id, 0xc) );
    if (flags & 0x01)   MyAlert(id, RSR_PRX);
    if (flags & 0x02)   MyAlert(id, RSR_CRC);
    if (flags & 0x04)   MyAlert(id, RSR_FAE);
    if (flags & 0x08)   MyAlert(id, RSR_FO);
    if (flags & 0x10)   MyAlert(id, RSR_MPA);
    if (flags & 0x20)   MyAlert(id, RSR_MUL);
    else                MyAlert(id, RSR_PHY);

    if ((flags = *RegAddr(id, 0x0d) ) != 0 && flags != 0x7f) {
        alignErrors[id] += flags;
        sprintf(st, "Slot %X, Packet %d: Frame alignment error tally (hex):%x\n", id +9, packetsSent-1, flags);
        Message(st);
    }
    if ((flags = *RegAddr(id, 0x0e)) != 0 && flags != 0x7f) {
        CRCErrorTally[id] += flags;
        sprintf(st, "Slot %X, Packet %d: CRC error tally:%x\n", id +9, packetsSent-1, flags);
        Message(st);
    }
    if ((flags = *RegAddr(id, 0x0f)) != 0 && flags != 0x7f) {
        missedPackets[id] += flags;
        sprintf(st, "Slot %X, Packet %d: Missed packet error tally:%x\n", id +9, packetsSent-1, flags);
        Message(st);
    }
    *(RegAddr(id, 0) ) = 0x22;  /* start, page 0 */
}

void
MyAlert(id, which)  /* informs operator of system errors and warnings */
{
    switch(which) {     /* transmission status */
        case NO_ERR:    return;
        case ISR_PTX:
```

```
            if (verbose) printf("Slot %X, packet %d:Interrupt status register: Successful transmit bit NOT set.\n", id+9, pa
            return;
        case TSR_PTX:
            if (verbose) Message("Transmit status register: Successful transmission bit NOT set.\n");
            return;
        case TSR_DFR:
            if (CRSMask) return;
            if (verbose) Message("Transmit status register: Non deferred transmission.\n");
            return;
        case TSR_CDH:
            if (CRSMask) return;
            ++heartFailures[id];
            if (verbose) Message("Transmit status register: Possible collision detect heartbeat failure.\n");
            return;
        case TSR_CRS:
            if (CRSMask) return;
            ++carrierLost[id];
            if (verbose) Message("Transmit status register: Carrier sense lost. Transmission not aborted.\n");
            return;
        case TSR_COL:
            if (CRSMask) return;
            ++collisions[id];
            if (verbose) Message("Transmit status register: Transmission collision detected.\n");
            return;
        case TSR_OWC:
            if (CRSMask) return;
            ++OWC[id];
            if (verbose) Message("Transmit status register: Out of window collision. Transmission not aborted.\n");
            return;
        case TSR_ABT:
            if (CRSMask) return;
            ++excessCollisions[id];
            if (verbose) Message("Transmit status register: Transmission aborted due to excessive collisions.\n");
            else    /*   printf("Transmit status register: Transmission aborted due to excessive collisions.\n");
            ignoreErr = -1;  happens rather often on a loaded network*/
            return;
        case ISR_TXE:
            if (CRSMask) return;
            if (verbose) {
                printf("Slot %X, Packet %d: ", id +9, packetsSent-1);
                Message("\nInterrupt status register: Transmit error (excessive collisions or FIFO overrun)\n");
            }
            else    /*   printf("Interrupt status register: Transmit error (excessive collisions or FIFO overrun)\n");
*/          return;

        case RSR_PRX:           /* reception status */
        case ISR_PRX:
            if (0) Message("Successful reception bit set.\n");
            return;
        case RSR_PHY:
            if (0) Message("Receive status register: Packet used a physical address.\n");
            return;
        case RSR_MUL:
            ++multicastsReceived[id];
            if (verbose) Message("Receive status register: Multicast address bit set.\n");
            return;
    }

    sprintf(st, "\nSlot %X, Packet %d: ", id +9, packetsSent-1);
    Message(st);                    /* report a serious error */
    switch(which) {
        case RAM_ERR:
            sprintf(st, "RAM failure at address %x\n", errorAddress);
            Message(st);
            break;
        case ISR_RXE:
            ++receiveErrors[id];
            Message("Interrupt status register: Receive error (CRC, Frame align, FIFO Overrun, or missed packet)\n");
            break;
        case ISR_OVW:
            ++bufferOverflows[id];
            Message("Interrupt status register: Overwrite warning (receive buffer is filled)\n");
            break;
        case ISR_CNT:
            Message("Interrupt status register: Counter overflow.\n");
            break;
        case TSR_FU:
            ++FIFOunderruns[id];
            Message("Transmit status register: FIFO underrun detected.\n");
            break;
        case RSR_CRC:
            ++CRCErrors[id];
            Message("Receive status register: Received packet with CRC error.\n");
            break;
        case RSR_FAE:
            ++alignErrors[id];
            Message("Receive status register: Frame alignment error detected.\n");
            break;
        case RSR_FO:
            ++FIFOoverruns[id];
            Message("Receive status register: FIFO overrun. Packet reception aborted.\n");
            break;
        case RSR_MPA:
            Message("Receive status register: Missed Packet due to buffer overflow.\n");
            break;
        default:
            sprintf(st, "Program alert ID = %d\n", which);  Message(st);
    }
```

```
    return;
}

QuitTest()
{
    FInfo    the_info;          /* used to set 'TEXT' type output file */

    PrintStats();
    return;                     /* for tool, results can be redirected using MPW commands */
    printf("Appending all results to file %s", fileName);
    if (GetFInfo(fileName, (short)0, &the_info) != noErr)   MyAlert(0, FINFO_ERR);
    the_info.fdType = 0x54455854;   /* 'TEXT' */
    if (SetFInfo(fileName, (short)0, &the_info) != noErr)   MyAlert(0, FINFO_ERR);
    fclose(outFile);
}

PrintStats()
{
    Message("\n");
    Message(TimeStamp() );
    sprintf(st, "\n\nTotal packets sent (base 10) = %d, ", packetsSent);
    Message(st);
    sprintf(st, "total packets received = %d\n", packetsReceived);
    Message(st);
    if (packetsSent) {
        sprintf(st, "Total retries = %d, Average Retries/Packet = %f\n",
                    Retries, (float)Retries/packetsSent);
        Message(st);
    }
    Message("Transmissions lost or corrupted-> Transmitter");
    PrintArray(lostOrCorrupt);
    Message("Excessive collisions (transmissions aborted)->");
    PrintArray(excessCollisions);
    Message("Transmission timeouts (transmissions aborted)->");
    PrintArray(transmitTimeouts);
    Message("Received CRC error tally register->");
    PrintArray(CRCErrorTally);
    Message("Packets received with CRC error bit set->");
    PrintArray(CRCErrors);
    Message("Packet frame alignment errors received->");
    PrintArray(alignErrors);
    Message("Buffer overflows received->");
    PrintArray(bufferOverflows);
    Message("Missed packets due to buffer overflow->");
    PrintArray(missedPackets);
    Message("FIFO underruns->");
    PrintArray(FIFOunderruns);
    Message("FIFO overruns->");
    PrintArray(FIFOoverruns);
    Message("Possible collision detect heartbeat failure bit set->");
    PrintArray(heartFailures);
    Message("Carrier sense lost (transmissions not aborted)->");
    PrintArray(carrierLost);
    Message("Total collisions (transmissions not aborted)->");
    PrintArray(collisions);
    Message("Out of window collisions (transmissions not aborted)->");
    PrintArray(OWC);
    Message("Multicasts detect flag set->");
    PrintArray(multicastsReceived);
}

PrintArray(theArray)
int theArray[];
{
    int i;

    Message("  Slot");
    for (i = 0; i < 6; i++)
    if (autoslots[i]) {
        sprintf(st, " %x: %d;",i+9, theArray[i]);
        Message(st);
    }
/*  putchar((char)0x08);        /* backspace one to remove last semicolon */
    Message("\n");
}

void
Query()
{
    char temp[64];

    printf(" Continue? (y/n)");
    gets(temp);
    if (temp[0] != 'y' && temp[0] != 'Y' && strlen(temp) > 0) {
        QuitTest();
        exit(0);
    }
    return;
}

char *
TimeStamp()             /* resolution to the second */
{
    static char lastTime[128];
    DateTimeRec d;

    GetTime(&d);
    sprintf(lastTime, "Date:%2d/%d/%2d  Time:%2d:%02d:%02d  ",
            d.month, d.day, d.year, d.hour, d.minute, d.second);
```

```
    return lastTime;
}

Message(string)         /* logs output to outputDirection devices */
char *string;
{

    static int firstTime = -1;

    if ((outputDirection & TO_FILE) && firstTime) {
        firstTime = 0;
        if (outFile = fopen(fileName, "a") ) {
            fputs("\n\n------------new test run---------------\n\n", outFile);
            Message(TimeStamp() );
        }
        else
            printf("error file %s could not be opened...", fileName);
    }

    if (outputDirection & TO_SCREEN)    printf(string);
    if ((outputDirection & TO_FILE) && outFile) {
        fputs(string, outFile);
    }
}


ZeroTallies()   /* reset error counters */
{
    short i;

    packetsSent = 0;
    Retries = 0;
    packetsReceived = 0;
    for (i=0; i < MAX_SLOTS; i++) {
        multicastsReceived[i] = 0;
        heartFailures[i] = 0;
        carrierLost[i] = 0;
        collisions[i] = 0;
        OWC[i] = 0;
        CRCErrorTally[i] = 0;
        CRCErrors[i] = 0;
        FIFOunderruns[i] = 0;
        FIFOoverruns[i] = 0;
        receiveErrors[i] = 0;
        alignErrors[i] = 0;
        excessCollisions[i] = 0;
        lengthErrors[i] = 0;
        missedPackets[i] = 0;
        bufferOverflows[i] = 0;
        transmitTimeouts[i] = 0;
        lostOrCorrupt[i] = 0;
    }
/*  Message("\n---Error counters reset to zero---\n"); */
}

#define ROM_START 0xF0000

TestROMRead(slot)       /* routine to test 3Com decode fix, called by Transmit via SendPacket */
{
    short i, j, *romPtr;

    romPtr = (short *) ( (slot+9)*0x100000 + ROM_START + 0x1A0); /* (0xsF01A0) = $ff01 or $ffff in ROM */
    for (i=0; i<30; ++i)
        j = *romPtr;

}

SlotDecodeRework()  /* routine to verify rework fix for Rev A to Rev C-J fix */
                    /* assumes slot(0-5) has already been initialized */
                    /* the routine SendPacket calls TestROMRead() while the packet is being sent */
{
    char    *xmitPtr, *rcvPtr;
    short   sender, receiver;
    short   myData = 0x0000;
    int     i, j;
    int     passed;

    printf("\nTesting for slot decode/data corruption rework…\n");
    for (sender=0; sender<6; ++sender) {
      if (!autoslots[sender])
        continue;
      passed = 0;                               /* card guilty until proven innocent */
      for (receiver=0; receiver<6; ++receiver) {
        if (!autoslots[receiver] || sender == receiver)
            continue;
        xmitPtr = (char *) RAMStart(sender);    /* point to transmit buffer in RAM */

        for (i=0; i<400; ++i)
            *(xmitPtr+i) = 0x00;                /* clear xmit buffer */

        for (i=0; i<100; ++i) {         /* unreworked cards fail approx.11% of the packets */
            if (SendPacket(sender, &(slotTable[receiver].ROMID), &myData, 94+i) ) {
                printf("Timeout: could not send rework test packet from %X to %X (check cable connections)\n",
                    sender+9, receiver+9);
                break;
            }
            if ((!BufferSize(receiver) || GetPacketLength(receiver) != 94+i) &&
                 !(*(RegAddr(sender, 4) ) & 0x08 ) ) {              /* not TSR abort */
                printf("Rework test packet from %X to %X corrupted (possible bad card)\n",
```

```
                    sender+9, receiver+9);
                break;
            }

            rcvPtr = (char *) RAMStart(receiver) + Boundry(receiver)*256;
            for (j=26; j<94+i; ++j) {
                if (*(rcvPtr+j)&&0xff != 0x00) {
/*                  printf("Expecting 0, found %x at location %lx, ",
                            *(rcvPtr+j)&&0xff, rcvPtr+j);
*/
                    break;
                }
            }
            while (BufferSize(receiver))    RemoveNextPacket(receiver);
            if (j != 94+i)  break;          /* failed test */
        }   /* for (i=0; i<100; i++)  */
        if (i == 100)       passed = -1;    /* completed every iteration */
      } /* for (receiver=0…) */
      if (passed)   printf("Slot %X has been reworked.\n", sender+9);
      else          printf("Slot %X may not have been reworked.\n", sender+9);
    }
    printf("Slot decode/data corruption rework test completed.\n\n");
}

pascal short GetAddr32(addr32, value)
    long    addr32;
    short   *value;
    extern;

FindCards()     /* locate EtherCards via block address (w/ & w/out using slot manager) */
{
    long    id;
    short   value;

    InstallMyErrV();                    /* set up bus error handler/find card routine */
    for (id = 0; id < MAX_SLOTS; ++id) {
        if (GetAddr32(0xf90f0000 + id*0x1000000, &value) == 0)
            printf("Slot %X: Empty or not responding.\n", id+9);
        else if ((value & 0xff00) == 0x0200) {
            GetAddr32(0xf90f0002 + id*0x1000000, &value);
            if ((value & 0xff00) == 0x6000) {
                autoslots[id] = -1;
                ++numCards;
/*              printf("Slot %X: EtherTalk card.\n", id+9); */
                if ( GetSlotMgrInfo(id+9) )     /* double check using slot manager */
                    fprintf(stderr, "Slot manager error: check or replace card %X's ROM…\n", id+9);
                else if (GetBoardID(id+9) != 8)
                    fprintf(stderr, "Slot manager error: board %X id not Apple/3Com id.\n", id+9);
            }
        }
    }
    InstallOldV();
}

PageSelectTest()        /* look for problems assoc. w/ strange address accesses */
{
    short   value;
    short   pSel;
    int     id;

    printf("\nStarting page select test…\n");
    InstallMyErrV();                    /* set up bus error handler/find card routine */
    for (id = 0; id < 6; ++id) if (autoslots[id]) {
        printf("Card %X: ", id+9);
        for (pSel = 0xF; pSel > 0xC ; --pSel) {
            if (GetAddr32(0xf9000000 + id*0x1000000 + pSel*0x10000, &value) == 0) {
                printf("Page %X: Unexpected bus error.\n", (int)pSel);
                break;
            }
        }
        if (pSel != 0xC) continue;
        for (pSel = 0xC; pSel >= 0 ; --pSel) {
            if (GetAddr32(0xf9000000 + id*0x1000000 + pSel*0x10000, &value) != 0) {
                printf("Page %X: Expected bus error, but returned %X.\n", (int)pSel, (int)value);
                break;
            }
        }
        if (pSel >= 0)  continue;
        else            printf("passed.\n");
    }
    InstallOldV();
    putchar('\n');
}

#define setPB(arg)  pB.spSlot = slot;\
                    pB.spID = arg;\
                    pB.spResult = (long) &pB

#define BoardId         32          /* board resoruce ID */
#define catBoard    1               /* used for defining Board sRsrc in Dir */

GetBoardID(slot)
int slot;           /* 9 to 0xe */
{
    SpBlock         pB;
    SInfoRecord     myInfoRec;

    pB.spSlot = slot;
    pB.spResult = (long) &myInfoRec;
```

```
    if (!SReadInfo(&pB))                /* Read ptr to directory of all resource types */

        if (myInfoRec.siInitStatusA != smEmptySlot) {

            setPB(catBoard);            /* Read ptr to board type resources */
            pB.spExtDev = 0;
            if (!SRsrcInfo(&pB))    {

                setPB(BoardId);         /* Read the board ID resource */
                if ( !SReadWord(&pB) )
                    return( (short) (pB.spResult & 0x0000ffff));
            }
        }
    return(0);                          /* Can't read the boardId */
}

int
GetSlotMgrInfo(slot)                /* get slot manager info & output error info */
int slot;           /* 9 to 0xe */
{
    SpBlock         pB;
    SInfoRecord     myInfoRec;
    OSErr           smErr;

    pB.spSlot = slot;
    pB.spResult = (long) &myInfoRec;
    smErr = SReadInfo(&pB);
    switch (myInfoRec.siInitStatusA)    {   /* Read ptr to directory of all resource types */
        case 0:                 break;  /* A-OK */
        case smEmptySlot:   fprintf(stderr, "Slot %X: Empty or not responding.\n", slot);   break;
        case smCRCFail:     fprintf(stderr, "Slot %X: CRC Check failed.\n", slot);      break;
        case smFormatErr:   fprintf(stderr, "Slot %X: declaration ROM format wrong.\n", slot);      break;
        case smRevisionErr: fprintf(stderr, "Slot %X: declaration ROM revision wrong.\n", slot);        break;
        case smNoDir:       fprintf(stderr, "Slot %X: no directory found.\n", slot);        break;
        case smLWTstBad:    fprintf(stderr, "Slot %X: ROM long word test failed.\n", slot);     break;
        case smNosInfoArray:fprintf(stderr, "Slot %X: SDM unable to allocate sInfo array memory.\n", slot); break;
        case smResrvErr:    fprintf(stderr, "Slot %X: declaration ROM reserved field used (fatal).\n", slot); break;
        case smUnExBusErr:  fprintf(stderr, "Slot %X: unexpected bus error occurred.\n", slot);     break;
        case smBLFieldBad:  fprintf(stderr, "Slot %X: ROM byte lanes field invalid.\n", slot);      break;
        case smFHBlockRdErr:fprintf(stderr, "Slot %X: F-header block couldn't be read.\n", slot);       break;
        case smDisposePErr: fprintf(stderr, "Slot %X: DisposePtr error occurred.\n", slot);     break;
        case smNoBoardsRsrc:fprintf(stderr, "Slot %X: No board slot resource.\n", slot);        break;
        case smGetPRErr:    fprintf(stderr, "Slot %X: sGetPRAMRec error occurred.\n", slot);        break;
        case smNoBoardId:   fprintf(stderr, "Slot %X: no board ID found.\n", slot);     break;
        case smInitTblErr:  fprintf(stderr, "Slot %X: error occurred initializing slot resource table.\n", slot);       breal
        case smNoJmpTbl:    fprintf(stderr, "Slot %X: can't create slot manager jump table.\n", slot);      break;
        case smBadBoardId:  fprintf(stderr, "Slot %X: board ID invalid.\n", slot);      break;
        case smInitStatVErr:fprintf(stderr, "Slot %X: ROM powerup/primary initialization code failed, returned %d.\n",
                            slot, myInfoRec.siInitStatusV);     break;
        default:
            fprintf(stderr, "Slot %X returned slot manager error #%d.\n", slot, myInfoRec.siInitStatusA);
    }
    return (myInfoRec.siInitStatusA);
}

BlastTCR(sID)
short sID;
{
    short i;
    long j;
    char *ptr;

    InstallMyErrV();                /* set up bus error handler/find card routine */
    ptr = (char *)(0x9E0000 + sID*0x100000);
    for (j = 0; j < 3000000; ++j) {
        i = *(ptr+0x28);
        *(ptr+0x24) = 0;
    }
    InstallOldV();
}

DataHoldTest(sID)
short sID;
{
    long i;
    char k;
    char j;
    char *ptr;

    printf("Slot %X data hold time test:", sID+9);
    *(RegAddr(sID, 0) ) = 0x62;                     /* page 1 */
    ptr = (char *)(0x9E0000 + sID*0x100000);

    for (i=0; i < 10000; ++i) {
        *(ptr+0x3C) = 0x62;
        if  (*(ptr+0x3C) != 0x62)
            continue;
        else break;
    }
    if (i > 0) {
        printf(" FAILED; %d iterations for setting page 1 reg.\n",
                sID+9, i);
        *(RegAddr(sID, 0) ) = 0x22;
        return;
    }

    for (j = 0; j < 0x7F; ++j) {
        *(ptr+0x10) = j;
        if ((k = *(ptr+0x10) ) != j) {
```

```
            printf(" FAILED; expected %X, found %x at address %X\n",
                    (int)j, (int)k, (int) (ptr+0x10) );
        *(RegAddr(sID, 0) ) = 0x22;
            return;
        }
    }
    *(RegAddr(sID, 0) ) = 0x22;
    printf("passed.\n");
}

pascal void BusErrDialog(codeLoc, mode, accessLoc)
long    codeLoc;
short   mode;               /* special status register from exception stack frame */
long    accessLoc;
{
    fprintf(stderr, "FAILURE: A bus error occurred executing code at or before $%lx\n", codeLoc);
    if (mode & 0x40)
            fprintf(stderr, "Program attempted a read to location $%lx\n", accessLoc);
    else    fprintf(stderr, "Program attempted a write to location $%lx\n", accessLoc);
    fprintf(stderr, "\nWARNING: Further MPW operations may now give erroneous results.\n");
    fprintf(stderr, "   Quit & retart MPW to continue.\n");
    fprintf(stderr, "DiagTool aborted…\n");
    InstallOldV();          /* remove bus error handler */
    exit(3);
}
```

```
/*  Ethercard verification routines and diagnostics in MPW C
    Mac interface routines contained in file MacEther.c
    © Apple Computer, Inc. 1987
    All rights reserved

    report bugs or useful modifications to Bill Weigel x-3898

    Mod History:
    20 May 87   BW  1st Rev.
    28 May 87   BW  Added Mac interface
    4 June 87   BW  Added ROMID field facilitating multiple test machines on
                    a single network (to test collision detection & retransmission)
    10 June 87  BW  Added verbose, reverse test direction commands
    12 June 87  BW  Added code to attempt to locate cause of dissapearing
                    packets on a heavily loaded network. not documented.
    18 April 88 BW  Added code to support ethernet packet analyzer

*/

#include <stdIO.h>
#include <Types.h>
#include <Events.h>
#include <Files.h>
#include <OSUtils.h>

#define min(A, B)   ((A) < (B)) ? (A) : (B)
#define TIME_ZIPS_BY -1
#define MAX_SLOTS 6
#define REENTER 0x1939

typedef struct {
    short l;            /* uses the low order byte of the unique ROM id */
    short m;
    short h;
} romID;
romID broadcastID = {
    0xffff,
    0xffff,
    0xffff
};
unsigned short triggerData = 0xABCD;        /* sent in a packet after error detected */
unsigned short regularData = 0x0000;        /* sent in a packet after no error detected */

enum statvals {SENT_PACK, RECV_PACK, TIMEOUT, WAITING};

struct slotinfo {
    romID ROMID;
    short slotID;           /* slot location (0-5, presently same as index) */
    char master;            /* master card if non-zero */
    enum statvals status;
    unsigned int lastReceptionNum;  /* Packet number of last valid reception */
    int timeout;            /* currently a loop decrementing counter */
}   slotTable[MAX_SLOTS];

#define TO_SCREEN 0x01
#define TO_FILE 0x02
#define TO_PRINT 0x04
short outputDirection = TO_SCREEN | TO_FILE;
FILE *outFile;              /* log errors to file */
char *fileName = "error.out";

void    MyAlert(),
        Query(),
        DumpRAM(),
        DumpRegs();

char    *RegAddr(),         /* these functions return ethernet registers */
        *RAMStart(),        /* and memory addresses */
        *memcpy1(),
        *GetSlotAddress();

char    *TimeStamp();

char    backspaces[] = "\010\010\010\010\010\010\010\010\010\010";

int ignoreErr;              /* temporarily disable error reporting mechanism */

unsigned int packetsSent;   /* frames sent successfully */
unsigned int Retries;       /* used to calculate Retries/Packet */
unsigned int packetsReceived;
unsigned int multicastsReceived[MAX_SLOTS];
unsigned int heartFailures[MAX_SLOTS];
unsigned int carrierLost[MAX_SLOTS];
unsigned int collisions[MAX_SLOTS];
unsigned int OWC[MAX_SLOTS];
unsigned int CRCErrorTally[MAX_SLOTS];
unsigned int CRCErrors[MAX_SLOTS];
unsigned int FIFOunderruns[MAX_SLOTS];
unsigned int FIFOoverruns[MAX_SLOTS];
unsigned int receiveErrors[MAX_SLOTS];
unsigned int alignErrors[MAX_SLOTS];
unsigned int excessCollisions[MAX_SLOTS];
unsigned int lengthErrors[MAX_SLOTS];
unsigned int missedPackets[MAX_SLOTS];
unsigned int bufferOverflows[MAX_SLOTS]; /* Packets lost due to lack of resources */
unsigned int transmitTimeouts[MAX_SLOTS];
unsigned int lostOrCorrupt[MAX_SLOTS];

char *errorAddress;         /* set by routine detecting the problem */
```

```
char st[256];
int numCards = 0;
int CRSMask;                /* mask carrier sense lost bit on receiving end */

extern short machnum;       /* these are set in MacEther.c */
extern short autoFlag;
extern short autoslots[];
extern short verbose;       /* when 0, only fatal errors are logged */
extern short reversed;      /* reverse transmit card to resolve error ambiguity */

/* pascal void SpinCursor(increment)
    short increment;
    extern;
*/

short sSlot;        /* valid range is presently 0 to 5 */
short mSlot;        /* master (sender) slot */
short pSlot;        /* promiscuous slot */
char *slotAddrs[6] =        /* address of 1st byte in each slot */
    {   (char *)0xf9900000, (char *)0xfaa00000, (char *)0xfbb00000,
        (char *)0xfcc00000, (char *)0xfdd00000, (char *)0xfee00000 };

pascal short InstallETInt(slot)
    short slot;
    extern;

pascal short RemoveETInt(slot)
    short slot;
    extern;

pascal void Debug()
    extern 0xa9ff;

dumpregisters()
{
    int i;

    if (!numCards) {
        SysBeep(3);
        return;
    }
    SetTest();
    for (i=0; i < MAX_SLOTS; i++)
        if (autoslots[i])
            DumpRegs(slotTable[i].slotID);
}

FindCards()     /* locate EtherCards via block address */
{
    int id;

    return;     /* or crash */
    for (id = 0; id < MAX_SLOTS; id++)
        printf("block id = %x %x %x", *(short *)(GetSlotAddress(id) + 0xf0000),
                            *(short *)(GetSlotAddress(id) + 0xf0002),
                            *(short *)(GetSlotAddress(id) + 0xf0004) );
}

SetTest()   /* set proper ids of cards to test */
{
    int i;

    numCards = 0;
    for (i=0; i < 6; i++) {
        if (autoslots[i]) {
            slotTable[i].slotID = i;    /* this relationship may change */
            numCards++;
        }
    }
}

AutoTest()
{
    char numbuffer[64];
    int i, oldVerbose, error;
    int seed = 512;
    short j;
    char *ptr;
    romID theID;

    if (autoFlag != REENTER) {      /* new test run */
        Message(" Ethercard verification diagnostic\n");
        Message(" Interrupt version 16 May 1988\n");

/*      printf("Promiscuous Slot (0-5 or -1 if none):");
        gets(numbuffer);    pSlot = atoi(numbuffer);
*/      pSlot = -1;
        if (pSlot != -1) {
            slotTable[pSlot].slotID = pSlot;
            InitSlot(pSlot);
        }
/*      else Message("No promiscuous slot…\n"); */
        Message("Single Card Test…");
        autoslots[3] = -1;                      /* test for slot C only */


        SetTest();      /* set proper ids of cards to test */
        ZeroTallies();
/*      if (numCards < 2) {
```

```
            Message("Loopback testing not implemented in this version\n");
            autoFlag = 0;
            Query();
            return;
        }
*/

        for (i = 0; i < MAX_SLOTS; i++)
            if (autoslots[i])
                InitSlot(slotTable[i].slotID);

        if (!(i = InstallETInt(0x0c) ) )
            printf("Install etest interrupt returned %d\n", i);
        else
            printf("etest interrupt installed successfully…\n");

        autoFlag = REENTER;
    }       /* if (autoFlag != REENTER) */
    do {
        while (1) {
            if (packetsSent<3)  SendPacket(3, &broadcastID, &triggerData, 940);
            else                Transmit(3);
/*          AnalyzeRegs(3); */
            if (!(packetsSent % 1000) ) {
                printf("%9d",packetsSent);
                printf(backspaces);
            }
            if (Button() )
                return;     /* do user interface stuff */
        }
        return;

        if (reversed)   GetNextPair(&sSlot, &mSlot); /* resolve transmit -    */
        else            GetNextPair(&mSlot, &sSlot); /* receive err ambiguity */
        if (++seed > 1500) {
/*          seed = 64; */
            seed = 1300;
            /* SpinCursor((short) 1); */
        }
        if (!(packetsSent % 1000) ) {
            printf("%9d",packetsSent);
            printf(backspaces);
        }

        if (pSlot != -1) {
            InitNIC(pSlot);
    /*      *(RegAddr(pSlot, 0x00) ) = 0x21;    /* stop NIC while changing regs */
    /*      *(RegAddr(pSlot, 0x0c) ) = 0x16;    /* Promiscuous,broad & multicast */
            *(RegAddr(pSlot, 0x00) ) = 0x61;    /* accept all multicasts */
            *(RegAddr(pSlot, 0x08) ) = 0xff;
            *(RegAddr(pSlot, 0x09) ) = 0xff;
            *(RegAddr(pSlot, 0x0a) ) = 0xff;
            *(RegAddr(pSlot, 0x0b) ) = 0xff;
            *(RegAddr(pSlot, 0x0c) ) = 0xff;
            *(RegAddr(pSlot, 0x0d) ) = 0xff;
            *(RegAddr(pSlot, 0x00) ) = 0x21;
            *(RegAddr(pSlot, 0x0c) ) = 0x20;
            *(RegAddr(pSlot, 0x00) ) = 0x22;        /* start promiscuous NIC */
            *(RegAddr(pSlot, 0x0c) ) = 0x16;
        }

        if (packetsSent < 6)        /* initialize xmit buffer */
            error = SendPacket(mSlot, &(slotTable[sSlot].ROMID), &regularData, seed);
        else
            error = Transmit(mSlot);

        if (error) {    /* timeout error */
            while (BufferSize(mSlot))   RemoveNextPacket(mSlot);
            while (BufferSize(sSlot))   RemoveNextPacket(sSlot);
            continue;
        }
        if (pSlot!=-1) *(RegAddr(pSlot, 0x00) ) = 0x21;     /* stop promiscuous NIC */

        if ((!BufferSize(sSlot) /* || GetPacketLength(sSlot) != seed */) &&
             !(*(RegAddr(mSlot, 4) ) & 0x08 ) ) {               /* not TSR abort */
/*          if (!SendPacket(sSlot, &broadcastID, &triggerData, 64) )    /* trigger analyser */
/*              --packetsSent;          /* don't count trigger packet in totals */
            ++lostOrCorrupt[mSlot];
            Message(TimeStamp() );
            sprintf(st, "\nPacket #%d was corrupted or lost ", packetsSent-1);
            Message(st);
            if (!BufferSize(sSlot) )                /* changed m to s 15 April 1988 */
                Message("(receive buffer empty)");
            else if (GetPacketLength(sSlot) != seed) /* changed m to s 15 April 1988 */
                Message("(data length field incorrect)");
            oldVerbose = verbose;
            verbose = -1;
            Message("\n----Transmitter packet header & data----\n");
            DumpRAM(mSlot, (short *)RAMStart(mSlot), 24);
            Message("----Transmitter Slot Status----\n");
            AnalyzeRegs(mSlot);
            Message("----Receiver Slot Status----");
            AnalyzeRegs(sSlot);

            if (pSlot!=-1) {
                Message("\n----Promiscuous Slot Status----");
                AnalyzeRegs(pSlot);
                *(RegAddr(pSlot, 0x00) ) = 0x21;    /* stop promiscuous NIC */
                Message("Into promiscuous DumpRAM\n");
```

```
                while (i = BufferSize(pSlot) ) {
                    printf("bndry = %x, current = %x", (int)Boundry(pSlot),
                            (int)Current(pSlot) );
                    GetPacketSource(pSlot, &theID);
                    printf("Buf siz = %x, source id = %x %x %x",
                        (int)BufferSize(pSlot), (int)theID.h, (int)theID.m, (int)theID.l);
                    DumpRAM(pSlot, Boundry(pSlot)*256, 8);
                    RemoveNextPacket(pSlot);
                    if (i == BufferSize(pSlot) )    /* bug fix */
                        break;
                }
            }

            verbose = oldVerbose;
            Message("\n\n");
            printf("%9d",packetsSent);  printf(backspaces);
/*          for (i = 0; i < MAX_SLOTS; i++)
                if (autoslots[i])
                    InitNIC(slotTable[i].slotID); /* hanging card fix */
        }
        CRSMask = 0;    AnalyzeRegs(mSlot); /* also updates error tally count */
        CRSMask = 1;    AnalyzeRegs(sSlot);
        while (BufferSize(mSlot))   RemoveNextPacket(mSlot);
        while (BufferSize(sSlot))   RemoveNextPacket(sSlot);

        /* if (pSlot != -1) AnalyzeRegs(pSlot); */
        if (pSlot != -1) {
            while (i = BufferSize(pSlot)) {
                printf("bndry = %x, current = %x", (int)Boundry(pSlot),
                        (int)Current(pSlot) );
                GetPacketSource(pSlot, &theID);
                printf("Buf size = %x, source id = %x %x %x\n",
                    (int)BufferSize(pSlot), (int)theID.h, (int)theID.m, (int)theID.l);
                DumpRAM(pSlot, Boundry(pSlot)*256, 8);
                RemoveNextPacket(pSlot);
                if (i == BufferSize(pSlot) )    /* bug fix */
                    break;
            }
        }
/*      printf("\n"); */
        ignoreErr = 0;  /* reset disable error reporting mechanism */
        if (Button() )
            return;     /* do user interface stuff */
    } while (TIME_ZIPS_BY);

    RemoveETInt(0x0c);
    QuitTest();
}

#define TPSR    0x00                    /* Transmit Page Start */
#define PSTART  (unsigned char)0x08     /* receive buffer start */
#define PSTOP   (unsigned char)0x1f

char *
GetSlotAddress(id)  /* returns slot address of id */
{
    return slotAddrs[id];
}

GetNextPair(mSlot, sSlot)
short *mSlot, *sSlot;
{
    static int lastMaster = -1;

    if (lastMaster < 0)
        while (!autoslots[++lastMaster] )
            ;
    *sSlot = slotTable[lastMaster].slotID;
    slotTable[lastMaster].master = 0;
    do {
        ++lastMaster;
        lastMaster %= MAX_SLOTS;
    } while (!autoslots[lastMaster] );

    *mSlot = slotTable[lastMaster].slotID;
    slotTable[lastMaster].master = -1;
}

int
SendPacket(sID, dROMID, data, dLength)          /* returns non-zero on failure */
short sID;
romID *dROMID;
short *data;     /* In this version, only 2 bytes copied to xmit buffer so as to maximize traffic */
short dLength;
{
    short *wPtr;

    wPtr = (short *) ((long)RAMStart(sID) & 0xfffffff0);
/*  *wPtr =       dROMID->h & 0x00ff;   /* mask extra bits in this version */
/*  *(wPtr + 1) = dROMID->m & 0xff00;
    *(wPtr + 2) = dROMID->l & 0xff00;   /* destination address */

    *wPtr =       0x1234;
    *(wPtr + 1) = 0x5678;
    *(wPtr + 2) = 0x9abc;   /* destination address */

    *(wPtr + 3) = slotTable[sID].ROMID.h & 0x00ff;
    *(wPtr + 4) = slotTable[sID].ROMID.m & 0xff00;
    *(wPtr + 5) = slotTable[sID].ROMID.l & 0xff00;  /* source address */
```

```
    *(wPtr + 6) = dLength;
    *(wPtr + 7) = (short)(packetsSent>>16);                 /* packet # data */
    *(wPtr + 8) = (short)(packetsSent & 0xffff);            /* packet # data */
    *(wPtr + 13) = *data;                                   /* test data */
    *(RegAddr(sID, 6) ) = (char) ((dLength+14) >> 8);       /* add header byte length */
    *(RegAddr(sID, 5) ) = (char) ((dLength+14) % 256);
    return Transmit(sID);
}

int
Transmit(id)
{
    int i;
    unsigned long tCount;
    long    *xmitComplete;

    xmitComplete = (long *) 0xa80;  /* last 4 bytes of AppScratch area */
    *xmitComplete = 0;              /* set non-zero by interrupt */
    tCount = TickCount() + 8*60;
    *(RegAddr(id, 7) )   = -1;      /* clear ISR bits */
/*  *(RegAddr(id, 0xf) ) = 0;       /* clear interrupt mask register */
    *(RegAddr(id, 0xf) ) = 0x0a;    /* set interrupt mask register transmit int only*/
    *(RegAddr(id, 0xd) ) = 0x00;    /* transmit config reg: normal operation */
    *(RegAddr(id, 0) )   = 0x22;    /* start NIC */
    *(RegAddr(id, 0) )   = 0x26;    /* & transmit */
    TestROMRead();                  /* test for decode address problem */
    for (i=0; i<2000; ++i)      ;   /* wait a bit before testing for status */
/*  while (!(*(RegAddr(id, 4) ) ) && tCount > (unsigned int)TickCount() )   /* was 7 */
    while (!(*xmitComplete) && tCount > (unsigned int)TickCount() )
        TestROMRead();                          /* wait for ISR status */
    if (tCount <= (unsigned int)TickCount() ) {
        Message(TimeStamp() );
        sprintf(st, "\nSlot %x, Packet #%d: NIC transmit timeout\n", id +9, packetsSent);
        Message(st);
        ++transmitTimeouts[id];
/*      for (i = 0; i < MAX_SLOTS; i++)
            if (autoslots[i])
                InitNIC(slotTable[i].slotID); /* hanging timeout fix */
        return -1;
    }
/*  else if (!( *(RegAddr(id, 7)) & 0x08) ) /* transmit error- excessive collisions or FIFO underrun */
        ++packetsSent;
    return 0;
}

int
EchoPacket(id)      /* send a packet back to its source */
short id;           /* returns non-zero on error */
{
    int cnt;
    romID theROMID;

    if (!BufferSize(id) ) {
        printf("EchoPacket called with no packets in buffer\n");
        return -1;
    }

    CopyData(id, RAMStart(id) + Boundry(id)*256 + 4, RAMStart(id), cnt);
    cnt = GetPacketLength(id);
    GetPacketSource(id, &theROMID);
    if (SendPacket(id, &theROMID, RAMStart(id)+16, cnt) )
        return -1;                  /* handle this error in main loop */
    RemoveNextPacket(id);           /* clean up buffer */
    return 0;
}

unsigned char
GetNextLink(id)         /* returns the next packet's page address */
{
    return *((unsigned char *)RAMStart(id) + Boundry(id)*256 + 1) & PSTOP;
}

RemoveNextPacket(id)    /* clear next packet from ring buffer without saving */
short id;
{
    unsigned char link;

    if (!BufferSize(id) ) {
/*      Message("RemoveNextPacket called with no packets in buffer\n"); */
        return -1;
    }
    link = GetNextLink(id);
    if (--link < PSTART) link = PSTOP; /* wraparound adjust */
/*  printf("removeNP slot %x link is %x\n", id+9, (int)link);   */
    ++packetsReceived;  /* should this be elsewhere??? */
    *(RegAddr(id, 3) ) = link;  /* reset NIC boundry pointer */
}

GetPacketLength(id)
{
    short *cnt;

    cnt = RAMStart(id) + Boundry(id)*256 + 16;  /* from receive buffer */
    return *cnt;
}

GetPacketSource(id, source)
short id;
romID *source;      /* returns the sender ROM id of next packet in memory */
```

```
{

    if (BufferSize(id) == 0) {      /* maybe call an alert??? */
        Message("Uh oh, GPS called with no packets in buffer...\n");
        return -1;
    }
    source->l = *(short *)(RAMStart(id) + Boundry(id)*256 + 0xe);
    source->m = *(short *)(RAMStart(id) + Boundry(id)*256 + 0xc);
    source->h = *(short *)(RAMStart(id) + Boundry(id)*256 + 0xa);
    return 0;
}

int
Boundry(id) /* returns page pointer to next package to remove from ring */
short id;
{
    unsigned char bPtr;

    if (id == pSlot)    *(RegAddr(id, 0) ) = 0x21;  /* page 0 */
    else                *(RegAddr(id, 0) ) = 0x22;
    bPtr = *((unsigned char *)RegAddr(id, 3) );
    if (++bPtr > PSTOP) /* ++bPtr points to next available buffer */
        bPtr = PSTART;
    return (int)bPtr;
}

int
Current(id) /* returns current page register (head ptr to receive ring buffer) */
short id;
{
    char curPtr;

    if (id == pSlot)    *(RegAddr(id, 0) ) = 0x61;  /* page 1 */
    else                *(RegAddr(id, 0) ) = 0x62;
    curPtr = *(RegAddr(id, 7) );
    if (id == pSlot)    *(RegAddr(id, 0) ) = 0x21;  /* page 0 */
    else                *(RegAddr(id, 0) ) = 0x22;
    return (int)curPtr;
}

int
BufferSize(id)  /* returns the page size of unread buffers */
short id;
{
    int num;

    num = Current(id) - Boundry(id);
    if (num < 0) num += PSTOP - PSTART;
    return num;
}

struct NICpair { /* each NICpair pair corresponds to data, register */
    char data;
    char reg;
} NICinit[] = {
    0x22, 0x0,  /* NIC off-line */
    0x21, 0x0,  /* init command register, abort DMA, NIC off-line, Page 0 */
    0x49, 0xE,  /* Data config set - word length DMA transfers */
    0x00, 0xA,  /*clear RBCR0 */
    0x00, 0xB,  /*clear RBCR1 */
    0x00, 0xC,  /* Rcv config set should be a menu item to set bits */
    0x02, 0xD,  /* NIC in loopback mode */
    PSTART,0x1, /* set PSTART reg - ring buffer init */
    PSTOP, 0x2, /* set PSTOP reg - ring buffer init */
    PSTOP, 0x3, /* set boundary reg - ring buffer init */
    TPSR, 0x4,  /* Xmit Page Start */
    0xFF, 0x7,  /* reset Interrupt Status register */
    0x00, 0xF,  /* set Interrupt Mask register */
/*  0x02, 0x6,  /* set Xmit byte count register 1 */
/*  0x00, 0x5,  /* set Xmit byte count register 0 */
    0x61, 0x0,  /* select page 1 registers */
    0x00, 0x1,  /* set Physical Address Regs to node addr - regs 1 -> 6 */
    0x00, 0x2, 0x00, 0x3, 0x00, 0x4, 0x00, 0x5, 0xFF, 0x6,
    0x00, 0x8,  /* set Multicast Address regs - regs 8 -> F */
    0x00, 0x9, 0x00, 0xA, 0x00, 0xB,
    0x00, 0xC, 0x00, 0xD, 0x00, 0xE, 0x00, 0xF,
    PSTART, 0x7,/* curr page ptr to ring buffer */
    0x21, 0x0,  /* Select Page 0 regs */
    0x22, 0x0,  /* set start mode */
    0x00, 0xD   /* initialize Xmit config register */
};

#define NO_ERR  0x00
#define RAM_ERR 0x01
#define OUT_OF_BOUNDS 0x02
#define ISR_RXE 0x03
#define ISR_TXE 0x04
#define ISR_OVW 0x05
#define ISR_CNT 0x06
#define TSR_COL 0x07
#define TSR_ABT 0x08
#define TSR_CRS 0x09
#define TSR_FU  0x0a
#define TSR_CDH 0x0b
#define TSR_OWC 0x0c
#define RSR_CRC 0x0d
#define RSR_FAE 0x0e
#define RSR_FO  0x0f
#define RSR_MPA 0x10
```

```
#define RSR_PHY 0x11
#define RSR_MUL 0x12
#define RSR_PRX 0x13
#define TSR_PTX 0x14
#define TSR_DFR 0x16
#define ISR_PTX 0x15
#define ISR_PRX 0x17
#define ISR_RDC 0x18

#define FINFO_ERR 0x25

#define NICOffset (char *)0xe003c

char *
RegAddr(id, n)      /* returns address of NIC register n in slot id */
short id, n;
{
    return slotAddrs[id] + NICOffset - n*4;
}

int
InitSlot(id)
short id;
{
    short i, j;

    *(RegAddr(id, 0) ) = 0x21;      /* turn off NIC during RAM test */
    MyAlert(id, InitRAM(id) );
    InitNIC(id);
    for (i=0; i < 8; i++)           /* the first 16 bytes */
        *((int *)RAMStart(id) + i) = 0; /* zero xmit buffer */
}

int
InitNIC(id)
short id;
{
    int i;

    for (i=0; i < sizeof(NICinit)/sizeof(struct NICpair); i++)
        *(RegAddr(id, NICinit[i].reg) ) = NICinit[i].data;

    *(RegAddr(id, 0) ) = 0x61;          /* Page 1 registers */
    slotTable[id].ROMID.h = *(short *)(GetSlotAddress(id) + 0xf0006);
    slotTable[id].ROMID.m = *(short *)(GetSlotAddress(id) + 0xf0008);
    slotTable[id].ROMID.l = *(short *)(GetSlotAddress(id) + 0xf000a);
    if (slotTable[id].ROMID.h == 0x700 && slotTable[id].ROMID.m == 0x900 &&
        slotTable[id].ROMID.l == 0xb00) {
        sprintf(st, "WARNING: card in slot %x may be missing ROM identifier chip.\n", id+9);
        Message(st);
        Message("Program cannot run without this ROM installed.\n");
        Query();
    }
    *((int *)RegAddr(id, 1) ) = 0;      /* multicast address has MSB set high */
    *((int *)RegAddr(id, 2) ) = ((int)slotTable[id].ROMID.h) << 16;
    *((int *)RegAddr(id, 3) ) = ((int)slotTable[id].ROMID.m) << 16;
    *((int *)RegAddr(id, 4) ) = 0;
    *((int *)RegAddr(id, 5) ) = ((int)slotTable[id].ROMID.l) << 16;
    *((int *)RegAddr(id, 6) ) = 0;

    slotTable[id].ROMID.h = (slotTable[id].ROMID.h >> 8) & 0x00ff;
                    /* so as not to be confused with a multicast address */
/*  slotTable[id].ROMID.m &= 0xff00;
    slotTable[id].ROMID.l &= 0xff00;
*/
                                        /* Physical Addr node # */
    *(RegAddr(id, 0) ) = 0x21;          /* Page 0 regs */
    i = *RegAddr(id, 0x0d);             /* clear tally counters */
    i = *RegAddr(id, 0x0e);
    i = *RegAddr(id, 0x0f);
/*  printf("block id = %x %x %x", *(short *)(GetSlotAddress(id) + 0xf0000),
                            *(short *)(GetSlotAddress(id) + 0xf0002),
                            *(short *)(GetSlotAddress(id) + 0xf0004) );
/*
    printf(" slot id = %x %x %x", *(short *)(GetSlotAddress(id) + 0xf0006),
                            *(short *)(GetSlotAddress(id) + 0xf0008),
                            *(short *)(GetSlotAddress(id) + 0xf000a) );
*/
    if (id != pSlot)    *(RegAddr(id, 0) ) = 0x22;          /* start NIC */
/*  sprintf(st, "slot %x: NIC initialized\n", id+9);
    Message(st); */
    return NO_ERR;
}

#define RAMOffset (char *) 0xD0000

char *
RAMStart(id)    /* return the first location of RAM on board in slot id (0-5) */
short id;
{
    return GetSlotAddress(id) + RAMOffset;
}

int
InitRAM(id) /* returns non-zero error on failure */
short id;
{
    register short *ptr;
    register short i;
```

```
    ptr = (short *) RAMStart(id);

    for (i = 0; i < 0x2000; i++)    /* check two bytes at a time */
        *(ptr+i) = 0xffff;
    for (i = 0; i < 0x2000; i++)
        if (*(ptr+i) & 0xffff != 0xffff) {  /* compiler returns 32 bit data */
            errorAddress = (char *) ptr + i;
            Message("1");
            return RAM_ERR;
        }

    for (i = 0; i < 0x2000; i++)
        *(ptr+i) = 0;
    for (i = 0; i < 0x2000; i++)
        if (*(ptr+i) != 0) {
            errorAddress = (char *) ptr + i;
            Message("2");
            return RAM_ERR;
        }

    for (i = 0; i < 0x2000; i++) {
        *(ptr+i) = i;
        if (*(ptr+i) != i) {
            errorAddress = (char *) ptr + i;
            Message("3");
            return RAM_ERR;
        }
    }


    for (i = 0; i < 0x2000; i++)
        *(ptr+i) = 0x5555;
    for (i = 0; i < 0x2000; i++)
        if (*(ptr+i) != 0x5555) {
            errorAddress = (char *) ptr + i;
            Message("4");
            return RAM_ERR;
        }

    for (i = 0; i < 0x2000; i++)
        *(ptr+i) = 0xaaaa;
    for (i = 0; i < 0x2000; i++)
        if (*(ptr+i) & 0xffff != 0xaaaa) {
            errorAddress = (char *) ptr + i;
            Message("5");
            return RAM_ERR;
        }


    Message("RAM Test completed \n");
    return NO_ERR;
}

void
DumpRegs(id)        /* of NIC in slot id (9 - e) */
short id;
{
    int i;
    char j;

    *(RegAddr(id, 0) ) = 0x21;
    sprintf(st,"NIC Register dump of machine %x, slot %x\nPage 0  ", machnum, id + 9);  Message(st);
    for (i=0; i < 16; i++) {
        if (i == 6)
            Message("6:XX ");   /* don't disturb data in FIFO register */
        else {
            j = *(RegAddr(id, i) );
            sprintf(st,"%1x:%02x ", i, (j & 0xff) );    Message(st);
        }
    }
    Message("\nPage 1  ");
    *(RegAddr(id, 0) ) = 0x61;
    for (i=0; i < 16; i++) {
        j = *(RegAddr(id, i) );
        sprintf(st, "%1x:%02x ", i, (j & 0xff) );   Message(st);
    }
    Message("\n");
    if (id != pSlot)    *(RegAddr(id, 0) ) = 0x21;
    else                *(RegAddr(id, 0) ) = 0x22;
}

void
DumpRAM(id, begin, cnt) /* display a range of memory in card id */
short id;
char *begin;
int cnt;
{
    begin = (RAMStart(id) + ((short)begin & 0x3ffe) ); /* start on word boundry */
    sprintf(st,"%4x: ", begin); Message(st);
    while (cnt-- > 0) {
        sprintf(st, "%04x ", *(short *)begin & 0xffff); Message(st);
        begin += 2;
        if (!((short)begin % 16) && cnt) {
            sprintf(st, "\n%4x: ", begin);
            Message(st);
        }
    }
    Message("\n");
```

```
    return;
}

char *memcpy1(to, from, cnt)
short *to, *from;
int cnt;
{
    cnt >>= 1;
    while (cnt--)
        *(to++) = *(from++);
    return from;
}

int
CopyData(id, from, to, cnt)
short id;
char *to, *from;    /* absolute addresses */
int cnt;            /* count in bytes */
{
    int first;      /* used to calculate page memory wraparound */

    first = min(RAMStart(id) + (PSTOP+1)*256 - from, cnt);
    memcpy1(to, from, first);
    to += first;
    if ( (cnt -= first) > 0)    memcpy1(to, RAMStart(id) + PSTART*256, cnt);
}

fillw(seed, to, cnt)    /* simple routine for filling a data buffer */
register int seed;
register short *to;
register int cnt;
{
    ++cnt;
    cnt >>= 1;          /* # of words in packet */
    while (cnt--)
        *(to++) = seed++;
    return;
}

void
swap_bytes(i)       /* ror.w #8, d0 */
register short *i;
{
    register short j;

    j = *i & 0xffff;
    *i = j<<8 | j>>8;
    return;
}

AnalyzeRegs(id)     /* check status registers of NIC for errors and such */
short id;
{
    register char flags;

    if (packetsSent < 6)
        return;         /* NIC may show false errors after power on */
/*  *(RegAddr(id, 2) ) = 0x21;  /* stop NIC while checking regs */
    flags = *(RegAddr(id, 7) );
    if (flags & 0x01)   MyAlert(id, ISR_PRX);
    if (!(flags & 0x02))    MyAlert(id, ISR_PTX);
    if (flags & 0x04)   MyAlert(id, ISR_RXE);
    if (flags & 0x08)   MyAlert(id, ISR_TXE);
    if (flags & 0x10)   MyAlert(id, ISR_OVW);
    if (flags & 0x20)   MyAlert(id, ISR_CNT);
    if (flags & 0x40)   MyAlert(id, ISR_RDC);

    flags = *(RegAddr(id, 4) );
    if (!(flags & 0x01))    MyAlert(id, TSR_PTX);
    if (flags & 0x02)   MyAlert(id, TSR_DFR);
    if (flags & 0x04) { MyAlert(id, TSR_COL); Retries += *(RegAddr(id, 5) ); }
    if (flags & 0x08)   MyAlert(id, TSR_ABT);
    if ((flags & 0x10) && !CRSMask) MyAlert(id, TSR_CRS);
    if (flags & 0x20)   MyAlert(id, TSR_FU);
    if (flags & 0x40)   MyAlert(id, TSR_CDH);
    if (flags & 0x80)   MyAlert(id, TSR_OWC);

    flags = *(RegAddr(id, 0xc) );
    if (flags & 0x01)   MyAlert(id, RSR_PRX);
    if (flags & 0x02)   MyAlert(id, RSR_CRC);
    if (flags & 0x04)   MyAlert(id, RSR_FAE);
    if (flags & 0x08)   MyAlert(id, RSR_FO);
    if (flags & 0x10)   MyAlert(id, RSR_MPA);
    if (flags & 0x20)   MyAlert(id, RSR_MUL);
    else                MyAlert(id, RSR_PHY);

    if ((flags = *RegAddr(id, 0x0d) ) != 0 && flags != 0x7f) {
        alignErrors[id] += flags;
        sprintf(st, "Slot %x, Packet %d: Frame alignment error tally (hex):%x\n", id +9, packetsSent, flags);
        Message(st);
    }
    if ((flags = *RegAddr(id, 0x0e)) != 0 && flags != 0x7f) {
        CRCErrorTally[id] += flags;
        sprintf(st, "Slot %x, Packet %d: CRC error tally:%x\n", id +9, packetsSent, flags);
        Message(st);
    }
    if ((flags = *RegAddr(id, 0x0f)) != 0 && flags != 0x7f) {
        missedPackets[id] += flags;
        sprintf(st, "Slot %x, Packet %d: Missed packet error tally:%x\n", id +9, packetsSent, flags);
```

```
        Message(st);
    }
    if (id != pSlot)
        *(RegAddr(id, 0) ) = 0x22;  /* start, page 0 */
}

void
MyAlert(id, which)  /* informs operator of system errors and warnings */
{
    switch(which) {     /* transmission status */
        case NO_ERR:    return;
        case ISR_PTX:
            if (verbose) Message("Interrupt status register: Successful transmission bit NOT set.\n");
            return;
        case TSR_PTX:
            if (verbose) Message("Transmit status register: Successful transmission bit NOT set.\n");
            return;
        case TSR_DFR:
            if (CRSMask) return;
            if (verbose) Message("Transmit status register: Non deferred transmission.\n");
            return;
        case TSR_CDH:
            if (CRSMask) return;
            ++heartFailures[id];
            if (verbose) Message("Transmit status register: Possible collision detect heartbeat failure.\n");
            return;
        case TSR_CRS:
            if (CRSMask) return;
            ++carrierLost[id];
            if (verbose) Message("Transmit status register: Carrier sense lost. Transmission not aborted.\n");
            return;
        case TSR_COL:
            if (CRSMask) return;
            ++collisions[id];
            if (verbose) Message("Transmit status register: Transmission collision detected.\n");
            return;
        case TSR_OWC:
            if (CRSMask) return;
            ++OWC[id];
            if (verbose) Message("Transmit status register: Out of window collision. Transmission not aborted.\n");
            return;
        case TSR_ABT:
            if (CRSMask) return;
            ++excessCollisions[id];
            if (verbose) Message("Transmit status register: Transmission aborted due to excessive collisions.\n");
            else    /*   printf("Transmit status register: Transmission aborted due to excessive collisions.\n");
            ignoreErr = -1;  happens rather often on a loaded network*/
            return;
        case ISR_TXE:
            if (CRSMask) return;
            if (verbose) {
                printf("Slot %x, Packet #%d: ", id +9, packetsSent);
                Message("\nInterrupt status register: Transmit error (excessive collisions or FIFO overrun)\n");
            }
            else    /*   printf("Interrupt status register: Transmit error (excessive collisions or FIFO overrun)\n");
*/          return;

        case RSR_PRX:           /* reception status */
        case ISR_PRX:
            if (0) Message("Successful reception bit set.\n");
            return;
        case RSR_PHY:
            if (0) Message("Receive status register: Packet used a physical address.\n");
            return;
        case RSR_MUL:
            ++multicastsReceived[id];
            if (verbose) Message("Receive status register: Multicast address bit set.\n");
            return;
    }

    sprintf(st, "\nSlot %x, Packet #%d: ", id +9, packetsSent);
    Message(st);                    /* report a serious error */
    switch(which) {
        case RAM_ERR:
            sprintf(st, "RAM memory failure at address %x\n", errorAddress);
            Message(st);
            break;
        case ISR_RXE:
            ++receiveErrors[id];
            Message("Interrupt status register: Receive error (CRC, Frame align, FIFO Overrun, or missed packet)\n");
            break;
        case ISR_OVW:
            ++bufferOverflows[id];
            Message("Interrupt status register: Overwrite warning (receive buffer is filled)\n");
            break;
        case ISR_CNT:
            Message("Interrupt status register: Counter overflow.\n");
            break;
        case TSR_FU:
            ++FIFOunderruns[id];
            Message("Transmit status register: FIFO underrun detected.\n");
            break;
        case RSR_CRC:
            ++CRCErrors[id];
            Message("Receive status register: Received packet with CRC error.\n");
            break;
        case RSR_FAE:
            ++alignErrors[id];
            Message("Receive status register: Frame alignment error detected.\n");
```

```
            break;
        case RSR_FO:
            ++FIFOoverruns[id];
            Message("Receive status register: FIFO overrun. Packet reception aborted.\n");
            break;
        case RSR_MPA:
            Message("Receive status register: Missed Packet due to buffer overflow.\n");
            break;
        default:
            sprintf(st, "Program alert ID = %d\n", which);  Message(st);
    }

    return;
}

QuitTest()
{
    FInfo    the_info;          /* used to set 'TEXT' type output file */

    PrintStats();
    printf("Appending all results to file %s", fileName);
    if (GetFInfo(fileName, (short)0, &the_info) != noErr)   MyAlert(0, FINFO_ERR);
    the_info.fdType = 0x54455854;   /* 'TEXT' */
    if (SetFInfo(fileName, (short)0, &the_info) != noErr)   MyAlert(0, FINFO_ERR);
    fclose(outFile);
}

PrintStats()
{
    Message("\n");
    Message(TimeStamp() );
    sprintf(st, "\n\nTotal packets sent (base 10) = %d, ", packetsSent);
    Message(st);
    sprintf(st, "total packets received = %d\n", packetsReceived);
    Message(st);
    if (packetsSent) {
        sprintf(st, "Total retries = %d, Average Retries/Packet = %f\n",
                Retries, (float)Retries/packetsSent);
        Message(st);
    }
    Message("Transmissions lost or corrupted-> Transmitter");
    PrintArray(lostOrCorrupt);
    Message("Excessive collisions (transmissions aborted)->");
    PrintArray(excessCollisions);
    Message("Transmission timeouts (transmissions aborted)->");
    PrintArray(transmitTimeouts);
    Message("Received CRC error tally register->");
    PrintArray(CRCErrorTally);
    Message("Packets received with CRC error bit set->");
    PrintArray(CRCErrors);
    Message("Packet frame alignment errors received->");
    PrintArray(alignErrors);
    Message("Buffer overflows received->");
    PrintArray(bufferOverflows);
    Message("Missed packets due to buffer overflow->");
    PrintArray(missedPackets);
    Message("FIFO underruns->");
    PrintArray(FIFOunderruns);
    Message("FIFO overruns->");
    PrintArray(FIFOoverruns);
    Message("Possible collision detect heartbeat failure bit set->");
    PrintArray(heartFailures);
    Message("Carrier sense lost (transmissions not aborted)->");
    PrintArray(carrierLost);
    Message("Total collisions (transmissions not aborted)->");
    PrintArray(collisions);
    Message("Out of window collisions (transmissions not aborted)->");
    PrintArray(OWC);
    Message("Multicasts detect flag set->");
    PrintArray(multicastsReceived);
}

PrintArray(theArray)
int theArray[];
{
    int i;

    Message(" Slot");
    for (i = 0; i < 6; i++)
    if (autoslots[i]) {
        sprintf(st, " %x: %d;",i+9, theArray[i]);
        Message(st);
    }
    putchar((char)0x08);
    Message("\n");
}

void
Query()
{
    char temp[64];

    printf(" Continue? (y/n)");
    gets(temp);
    if (temp[0] != 'y' && temp[0] != 'Y' && strlen(temp) > 0) {
        QuitTest();
        exit(0);
    }
    return;
```

```
}

char *
TimeStamp()             /* resolution to the second */
{
    static char lastTime[128];
    DateTimeRec d;

    GetTime(&d);
    sprintf(lastTime, "Date:%2d/%d/%2d  Time:%2d:%02d:%02d  ",
            d.month, d.day, d.year, d.hour, d.minute, d.second);
    return lastTime;
}

Message(string)         /* logs output to outputDirection devices */
char *string;
{

    static int firstTime = -1;

    if ((outputDirection & TO_FILE) && firstTime) {
        firstTime = 0;
        if (outFile = fopen(fileName, "a") ) {
            fputs("\n\n------------new test run---------------\n\n", outFile);
            Message(TimeStamp() );
        }
        else
            printf("error file %s could not be opened...", fileName);
    }

    if (outputDirection & TO_SCREEN)    printf(string);
    if ((outputDirection & TO_FILE) && outFile) {
        fputs(string, outFile);
    }
}


ZeroTallies()   /* reset error counters */
{
    short i;

    packetsSent = 0;
    Retries = 0;
    packetsReceived = 0;
    for (i=0; i < MAX_SLOTS; i++) {
        multicastsReceived[i] = 0;
        heartFailures[i] = 0;
        carrierLost[i] = 0;
        collisions[i] = 0;
        OWC[i] = 0;
        CRCErrorTally[i] = 0;
        CRCErrors[i] = 0;
        FIFOunderruns[i] = 0;
        FIFOoverruns[i] = 0;
        receiveErrors[i] = 0;
        alignErrors[i] = 0;
        excessCollisions[i] = 0;
        lengthErrors[i] = 0;
        missedPackets[i] = 0;
        bufferOverflows[i] = 0;
        transmitTimeouts[i] = 0;
        lostOrCorrupt[i] = 0;
    }
    Message("\n---Error counters reset to zero---\n");
}

#define true32b 1
#define false32b 0

pascal void SwapMMUMode(c)
    char *c;
    extern 0xA05D;

TestROMRead()       /* routine to test 3Com fix */
{
    char c, mode, *aPtr;

    return;
    aPtr = (char *) 0xFEFF0000;     /* point to 1st byte in ROM of Caliente card slot E */
    mode = true32b;
    SwapMMUMode(&mode);
/*  c = *aPtr;                      /* 32 bit read from NuBus address */
    mode = false32b;
    SwapMMUMode(&mode);
}

TestRead()      /* routine to time 3Com fix */
{
    char c, mode;
    short *aPtr;
    int i;

    aPtr = (short *) 0xD00000;      /* point to 1st byte in ROM of Caliente card slot D */
    *aPtr = 0xabcd;                     /* 32 bit read from NuBus address */
    for (i=0; i<1500; ++i)      ;   /* wait a bit before testing for status */
    *aPtr = 0xabcd;                     /* 32 bit read from NuBus address */
}
```