IBM Personal Computer - Ethernet (IE) Controller/Transceiver

External Reference Specifications

3Com Corporation

March 15, 1983

IBM Personal Computer - Ethernet (IE)
Controller/Transceiver

External Reference Specifications

3Com Corporation

1. Introduction

The IE is a low cost Ethernet controller/transceiver for IBM Personal Computers and conforms to the Ethernet Specification, Version 1.0, 30 September 1980, as published by DEC, Intel and Xerox. With one minor exception, it implements levels one and two of the Open Systems Interconnect Model of the International Standards Organization.

Level One Functions, Physical Layer

- Coax/station electrical isolation.
- Bit transmission/reception.
- Carrier sense.
- Transmit collision detection.
- Encoding/decoding.
- Preamble generation/removal

Level Two Functions, Data Link Layer

- Frame check sequence generation/checking.
- Carrier deference.
- Transmit collision enforcement.
- Collision fragment (runt) filtering.
- Bad packet filtering.
- Address recognition.

The controller on the IE incorporates a VLSI Ethernet Data Link Controller, the Seeq 8001 or EDLC, and a single, two Kbyte, packet buffer designed to operate with the 8237A DMA controller found on the IBM System Board. It also has provisions for external loopback and can use one of the interrupt channels of the 8259A interrupt controller on the system board. In addition to a conventional Ethernet controller, the IE contains an Ethernet transceiver providing complete Level one and two functionality on one printed circuit card.

2. Architecture

The IE has a single two Kbyte packet buffer (large enough for the longest Ethernet packet) shared between transmit and receive. Once a packet is transferred to the buffer for transmission and a transmit

initiated, the software must intervene only in case of a collision. To receive a packet, software selects one of several address recognition modes; when a suitable packet arrives, the controller places it in the packet buffer. When enabled for multicast recognition, address screening of multicast packets must be performed by the system software.

Programs address the IE through a block of sixteen registers in the I/O space of the 8088. These registers are used to write commands, read status information, and access packet data. In contrast with IBM's practice, the IE's base address is jumper settable (32 values). The IE also has a four Kbyte ROM for program storage. The base address of the ROM in memory space is also jumper settable (32 values).

Access to the packet buffer switches between the system bus and the Ethernet under program control. Ethernet access can be receive, transmit with automatic rollover to receive, or loopback.

One of the I/O registers provides a one byte window on the packet buffer. Another register, GP, the general purpose buffer pointer, holds the address of the byte visible through the packet buffer window. Reading and writing the window automatically increment GP permitting sequential access to the packet buffer from the system bus. Writing GP then reading or writing the window gives the effect of random access.

The packet buffer can be loaded and unloaded using the 8237A DMA controller on the system board to repetitively read or write the window. The IE can request DMA service, detect the end of the transfer and interrupt when the DMA is done.

All interrupts go through the 8259A interrupt controller on the system board. Each type of IE interrupt is enabled independently of other types; however, the IE has only one interrupt line to the system. This arrangement requires software to scan the IE status registers to determine the cause of an interrupt. Consistent with IBM PC practice, both DMA service request and interrupt request line drivers can be disabled.

3. System Interface

The IE has two sixteen bit buffer address pointers (registers), GP and the receive buffer pointer, RP. RP is used only by the receive side of the controller for packet reception. The transmit side of the Ethernet logic and the system bus use GP for access to the packet buffer. Loopback operation uses both.

Transmit packets are end-aligned within the packet buffer. Software uses GP when filling the buffer. Before transmitting the buffer, software reloads GP so that it points to the first byte of the packet.

Receive packets are front-aligned in the buffer. Therefore, after the

IE receives a packet, RP contains the packet length in bytes. If the packet length exceeds the legal maximum, the first 2048 bytes will be saved in the buffer. After the 2048th byte, RP locks up preventing any buffer overwrite; reading a packet length of 2048 (800 hex) from RP indicates a packet of at least 2048 bytes.

During loopback, the controller reads the end-aligned transmit packet from the buffer and writes the received packet front-aligned in the buffer. A maximum length packet can be looped back. The received packet may overwrite part of the transmitted packet. Loopback requires the IE to be connected to an Ethernet by the onboard or external transceiver, or fitted with special BNC or DA-15 loopback plugs by the user.

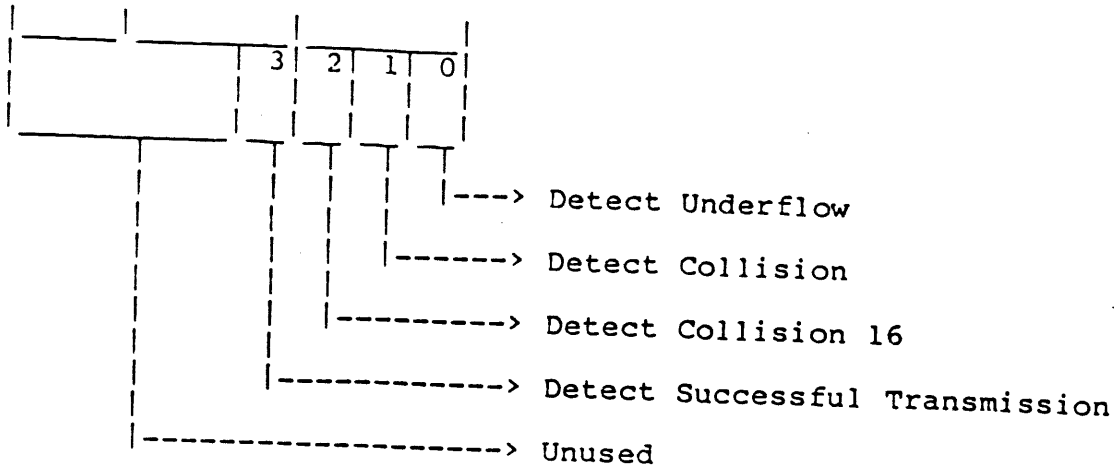
The IE's Ethernet station address is stored in a PROM, whose contents are accessible through another one byte window register similar to the window register used to access the packet buffer. Software uses GP to address the station address PROM available in locations zero thru five with station address byte zero at address zero. Unlike access to the packet buffer, reading the station address does NOT auto-increment GP; software must explicitly increment it.

The IE provides two sets of registers for the Ethernet station address. One set is read only; the other set is write only. Software must program the station address by setting the write only register set. The station address provided in the PROM serves only to provide a "hint" about what the station address should be.

3.1. IE Controller Register Map

	READ	WRITE
0		Station Addr 0
1		Station Addr 1
2		Station Addr 2
3		Station Addr 3
4		Station Addr 4
5		Station Addr 5
6	Receive Status	Receive Command
7	Transmit Status	Transmit Command
8	GP Buffer Pointer [LSB]	GP Buffer Pointer [LSB]
9	GP Buffer Pointer [MSB]	GP Buffer Pointer [MSB]
A	RCV Buffer Pointer [LSB]	RCV Buffer Pointer Clear
B	RCV Buffer Pointer [MSB]	
C	Ethernet Address Prom Window	
D		
E	Auxiliary Status	Auxiliary Command
F	Buffer Window	Buffer Window

3.2. Transmit Command Register



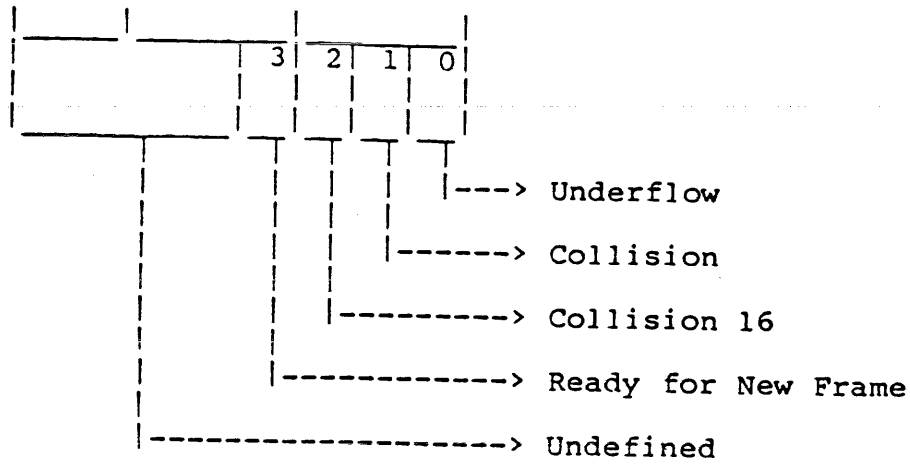
A packet transmission can terminate for any of four different reasons: successful transmission, collision, sixteenth successive collision without successful transmission, and underflow. After each collision, data remains in the packet buffer undisturbed; but, software must reset GP and explicitly restart the transmission; once restarted, the IE delays the appropriate amount of time before actually retransmitting the packet. After the sixteenth consecutive collision further attempts to retransmit should be abandoned on the assumption that the network is overloaded or has failed.

Underflow occurs only when transmitting packets without valid FCS and should not be seen during routine operation of the controller.

Software can choose whether to ignore or detect any of the four conditions listed above. A one in the corresponding bit position detects the condition; a zero ignores the condition. Detecting a condition is not sufficient to generate an interrupt. In order for the IE to generate interrupts, software must also set Request Interrupt and DMA Enable.

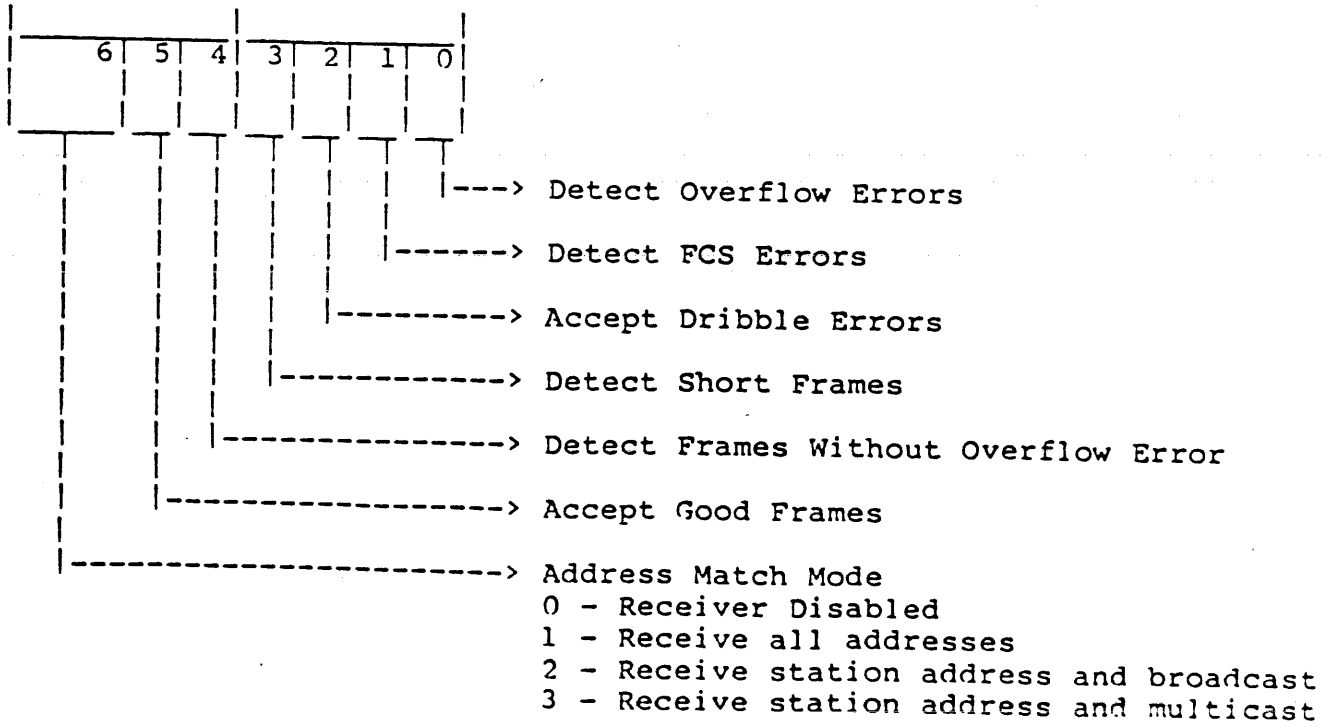
See the description of the Auxiliary Command Register below for details concerning underflow and interrupts.

3.3. Transmit Status Register



The controller loads the transmit status register only after each transmission or attempted transmission. If interrupts are enabled for transmit, reading the status register clears the interrupt.

3.4. Receive Command Register

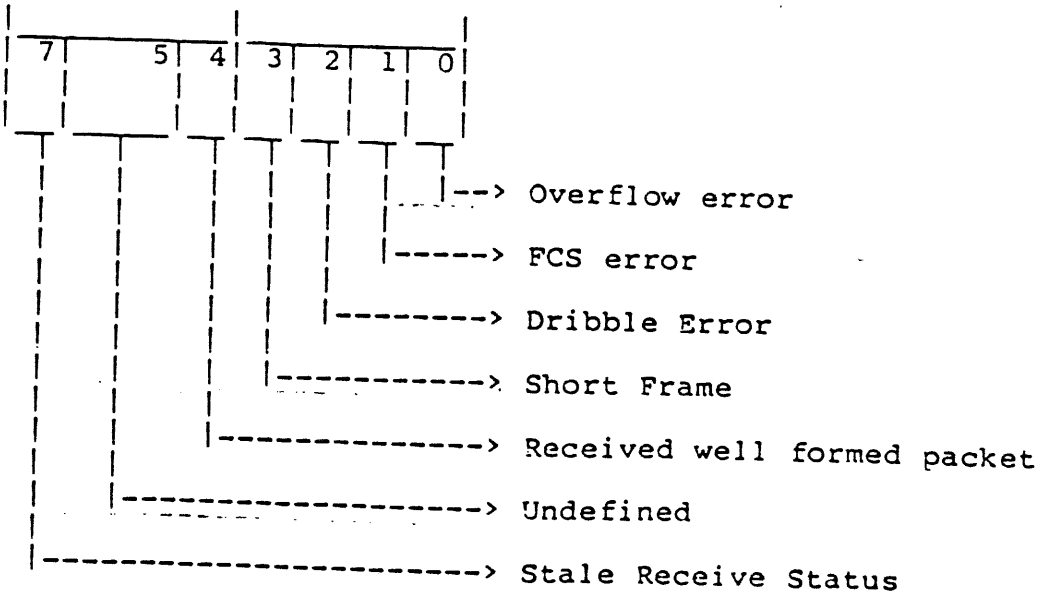


Software can program the IE to detect only certain classes of packets; all unwanted packets are discarded without intervention. The Address Match Mode controls whether to accept packets by examining their destination addresses. If the match mode is zero, the IE will not detect any packet; mode one accepts packets regardless of the contents of

their destination addresses. Modes two and three compare the destination of each packet with the station address registers stored in IE registers zero through five.

Other bits in the command register allow software to further qualify packets before detecting them. The IE accepts only well formed packets (legal size, no FCS error, and no overflow); to receive well formed packets, software must set the Accept Good Frame bit. Dribble indicates the packet did not end on a byte boundary; there were a few extra bits after the last byte. The controller will accept packets with dribble errors as long as the packet does not have anything else wrong with it and software sets Accept Dribble Errors. All other bits are useful only to detect packets with errors. The controller will not accept packets with errors; but, software can detect them in order to keep counts for diagnostic purposes. Short frames are packets whose length is less than 60 bytes, excluding preamble and FCS; these are probably collision fragments. FCS error means the four byte FCS computed on receipt did not match the FCS in the packet. (Note that an Ethernet "alignment error" is equivalent to a packet with dribble and FCS errors.) Overflow errors happen when the controller tries to accept a packet, but, the packet buffer is not available. The buffer might already have a packet, or the buffer might belong to the system bus rather than the Ethernet.

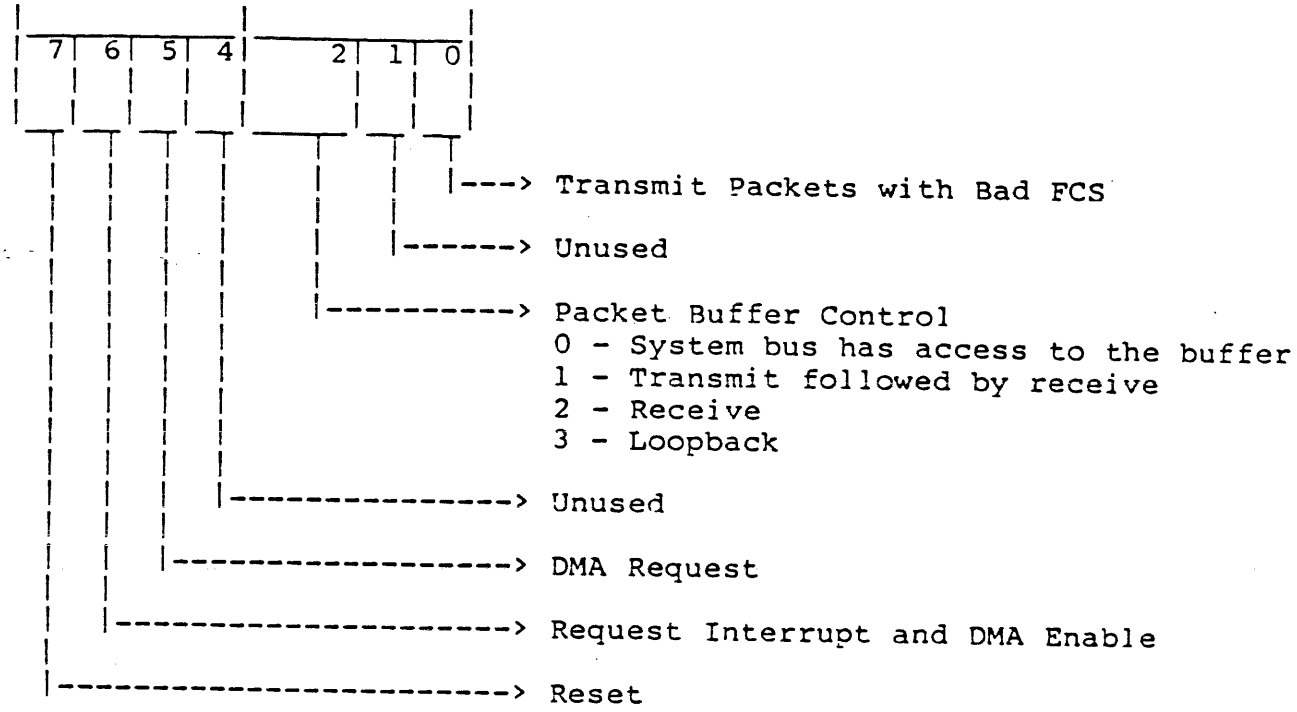
3.5. Receive Status Register



Software defines the class of "interesting" packets by setting the receive command register. The controller changes the status register after any packet goes by on the network whether or not it was interesting. If the controller detects an interesting packet, the Stale Receive Status goes to zero; once the Stale Receive Status is zero the controller discards all packets until software reads the status register; this guarantees that software reads the status associated with the detected packet. Reading the status register sets the

Stale Receive Status back to one; the IE can then detect the next interesting packet that comes by on the Ethernet. If receive interrupts are enabled, reading the status register also clears the interrupt.

3.6. Auxiliary Command Register



Writing a one in Reset, resets all control and status registers in the IE. Software must explicitly set this bit to zero after setting it to one; leaving Reset on has the effect of perpetually resetting the controller.

Request Interrupt and DMA Enable, RIDE, permits the IE to drive both the interrupt request, IRQ, and DMA service request, DRQ, signals on the system bus. Jumpers on the IE card select DMA channel (either one or three), and interrupt channel (either three or five). When RIDE is zero, the IE cannot generate interrupts or DMA transfers. Bits in the transmit and receive command registers can be set to detect certain conditions; however, no interrupts can result until RIDE is a one.

Software must manipulate RIDE with care. When RIDE is zero the state of the associated IRQ and DRQ lines on the system bus can be undefined. Leaving these lines in an undefined state when their associated DMA and interrupt channels are active can result in strange and unpredictable behavior. Software must insure that the associated IRQ and DRQ lines are not used by other peripheral devices before setting RIDE to one. Neither setting of RIDE is safe under all circumstances!

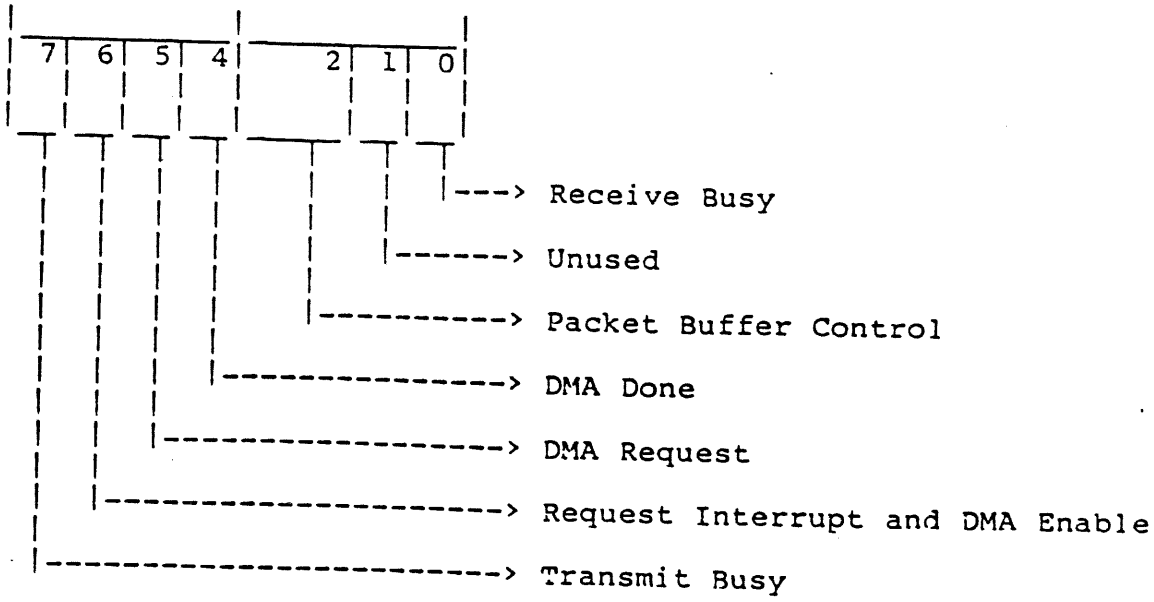
Setting DMA Request to one starts a DMA transfer. The IE interrupts at the completion of the transfer. Setting DMA Request to zero,

disables DMA Service request, clears DMA Done, and clears the interrupt.

Bits two and three of the auxiliary command register control access to the packet buffer. If both bits are zero, the buffer "belongs" to the system bus; software is free to read and write the buffer without interference from the Ethernet. If either of the bits are not zero the packet buffer belongs to the Ethernet. If bit 3 is one, the controller will accept one packet from the network. The setting of the receive command register can discard certain classes of packets and guarantee that only "interesting" packets are detected and reach the packet buffer. After receiving a packet, the controller leaves the size, excluding preamble and FCS, in bytes, in RP. If bit 2 goes to one, the controller transmits the packet buffer. If both bits are one, the controller transmits the packet buffer and simultaneously receives from the Ethernet writing the packet back into the beginning packet buffer.

Setting the low order bit of the auxiliary command register to one causes the IE to transmit packets with bad FCS. This bit is useful for testing the receive FCS circuitry.

3.7. Auxiliary Status Register



Software starts a DMA transfer by programming the proper channel of the 8259A DMA controller on the system board and setting DMA Request to one on the IE. When the DMA transfer ends, DMA Done goes to one; software clears DMA Done by setting DMA Request to zero.

Receive Busy goes to one whenever the controller is armed to receive a packet; this happens implicitly after transmitting a packet, or, explicitly by setting the Packet Buffer Control to receive or loopback. Receive Busy goes to zero after the controller accepts a packet. Software must wait 800 nanoseconds after receive busy goes to

zero before reading the receive status register.

Transmit Busy is meaningful only when the packet buffer control is set for loopback or transmit; while the packet buffer is switched to the bus or is in receive mode Transmit Busy will be set. Transmit Busy remains at one when software starts a transmit by setting the Packet Buffer Control to one. Transmit Busy goes to zero upon a collision or a successful transmission. Software can distinguish between these two cases by examining the transmit status register. Switching the packet buffer back to the bus sets Transmit Busy back to one.

4. IE Programming

To transmit a packet, first set the Packet Buffer Control to zero; this gives the system bus access to the buffer. Load the packet into the buffer so that the last byte of the packet coincides with the last byte of the packet buffer. Load GP so that it points to the first byte of the packet in the buffer. Start the transmission by setting the Packet Buffer Control to one. The transmission terminates when Transmit Busy goes to zero; read the transmit status register to determine whether there was a collision or a successful transmission.

In case of collision, set the Packet Buffer Control to zero, reload GP, and set the Packet Buffer Control to one; this retransmits the packet. Again wait for Transmit Busy to go to zero; then, read the transmit status register to determine why the transmission terminated.

Receiving packets requires both one time initialization of the controller and manipulation of the IE for each packet that arrives. The one time initialization includes reading the station address PROM, loading the station address registers, and setting the receive command register. In the programming example the routines "getaddr" and "setaddr", read the station address PROM and write the station address registers respectively.

To receive a packet clear RP and set the Packet Buffer Control to two; this initializes the read pointer and gives the packet buffer to the Ethernet. 800 nanoseconds after Receive Busy goes to zero, the receive status register has the status of the packet just received. The size of the packet, in bytes, is in RP. Software must set the Packet Buffer Control to zero before reading the packet from the buffer.

The following code, written in C, initializes the controller, transmits a single packet of 1000 bytes, and then receives well formed broadcast and packets addressed only to the station. The main program is found at the end of the example. The routines "inb", "inw", "outb", and "outw" read and write words and bytes on the IBM PC's I/O bus; the routine "inbs" reads a byte sign extended into a word. All of the examples are polled I/O; no use is made of the interrupt circuitry.

```
/* the various IE command registers */
#define IE(num)          (0x300+0x10*num)
```

```
#define EDLC_ADDR(num)    (num)              /* EDLC station address, 6 bytes */
#define EDLC_RCV(num)     ((num)+0x6)        /* EDLC receive command and stat */
#define EDLC_XMT(num)     ((num)+0x7)        /* EDLC transmit command and status */
#define IE_GP(num)   ((num)+0x8)     /* transmit, station address PROM bp */
#define IE_RP(num)   ((num)+0xa)     /* receive buffer pointer */
#define IE_SAPROM(num)    ((num)+0xc)     /* 1 byte window on station address */
#define IE_CSR(num) ((num)+0xe)      /* IE command and status */
#define IE_BFR(num) ((num)+0xf)      /* 1 byte window on packet buffer */

/* bits in EDLC_RCV, interrupt enable on write, status when read */
#define EDLC_NONE    0x00 /* match mode in bits 5-6, write only */
#define EDLC_ALL     0x40 /* promiscuous receive, write only */
#define EDLC_BROAD   0x80 /* station address plus broadcast */
#define EDLC_MULTI   0xc0 /* station address plus multicast */

#define EDLC_STALE   0x80 /* receive CSR status previously read */
#define EDLC_GOOD         0x20    /* well formed packets only */
#define EDLC_ANY     0x10 /* any packet, even those with errors */
#define EDLC_SHORT   0x08 /* short frame */
#define EDLC_DRIBBLE      0x04 /* dribble error */
#define EDLC_FCS     0x02 /* CRC error */
#define EDLC_OVER    0x01 /* data overflow */

#define EDLC_RERROR (EDLC_SHORT|EDLC_DRIBBLE|EDLC_FCS|EDLC_OVER)
#define EDLC_RMASK  (EDLC_GOOD|EDLC_ANY|EDLC_RERROR)

/* bits in EDLC_XMT, interrupt enable on write, status when read */
#define EDLC_IDLE    0x08 /* transmit idle */
#define EDLC_16           0x04 /* packet experienced 16 collisions */
#define EDLC_JAM     0x02 /* packet experienced a collision */
#define EDLC_UNDER   0x01 /* data underflow */

/* bits in IE_CSR */
#define IE_RESET     0x80 /* reset the controller */
#define IE_RIDE           0x40 /* request interrupt/DMA enable */
#define   IE_DMA          0x20 /* DMA request */
#define IE_EDMA           0x10 /* DMA done */

#define IE_LOOP           0x0c /* 2 bit field in bits 2 and 3, loopback */
#define IE_RCVEDLC   0x08 /* gives buffer to receive */
#define   IE_XMTEDLC      0x04 /* gives buffer to transmit */
#define IE_SYSBFR    0x00 /* gives buffer to processor */

#define IE_CRC       0x02 /* causes CRC error on transmit */
#define IE_RCVBSY    0x01 /* receive in progress */

/* miscellaneous sizes */
#define BFRSIZ 0x800          /* number of bytes in a buffer */
#define RUNT      60          /* smallest legal size packet, no fcs */
#define GIANT   1514          /* largest legal size packet, no fcs */

error(s) char *s; {printf("%s ", s);}

/* call DOS to test for keystroke, if its ^C get back to DOS */
```

```
sense_key() {char c;
     if (dos(0xb)) {if ((c = dos(8)&0177)==3) exit(0); return(c);}
     else return(0);}

/* low level transmit routines */
iereset(base) short base; {
     outb(0xa, 5);              /* turn off DMA channel 1 */
     outb(0x21, 0xa0);          /* mask interrupt level 5 */
     outb(IE_CSR(base), IE_RESET); /* reset them */
     outb(IE_CSR(base), 0);
     if (inb(IE_CSR(base)) != 0x80 ) error("Can't reset IE_CSR");
     if ((inb(EDLC_XMT(base))&0x0f) != 0) error("Can't clear EDLC_XMT");
     if ((inb(EDLC_RCV(base))&0x9f) != EDLC_STALE)
          error("Can't reset EDLC_RCV");}

xmt_start(base, size) short base; int size; {
     char c= inb(EDLC_XMT(base));
     outb(IE_CSR(base), IE_RIDE); /* make sure its out of transmit mode */
     outw(IE_GP(base), BFRSIZ-size);    /* before zapping counter */
     outb(EDLC_XMT(base), EDLC_16|EDLC_JAM|EDLC_UNDER|EDLC_IDLE);
     outb(IE_CSR(base), IE_RIDE|IE_XMTEDLC);}

/* returns     0 for successful transmit
          1 timed out
          2 collision
          3 data underflow
          4 idle not set after transmit
          5 16 collisions */
xmt_wait(base, size, stall) short base; int stall, size; {
     int i; char c;
     if ( (inb(IE_CSR(base))&IE_XMTEDLC) == 0)
          error("buffer not switched to transmit, xmt_wait");
     i = stall;
     do {
          if ( inbs(IE_CSR(base)) < 0 ) continue;
          c = inb(EDLC_XMT(base));
          if (c&EDLC_UNDER) return(3);  /* underflow */
          if (c&EDLC_16)    return(5);  /* 16 successive collisions? */
          if (c&EDLC_JAM)   return(2);  /* collision? */
          if (inw(IE_GP(base))==0x800) {
               if (!(inb(EDLC_XMT(base)) & EDLC_IDLE)) return(4);
               return(0);}}
     while(i-->=0);
     return(1);}

retransmit(base, mode, size) short base, mode, size; {
     int i = 0, org = BFRSIZ-size, k;
     if ( (inb(IE_CSR(base))&IE_XMTEDLC) == 0)
     error("buffer not switched to transmit, retransmit");
     while ( inbs(IE_CSR(base)) < 0 )
          if (++i > 1000) {error("retransmit timed out"); break;}
     outb(IE_CSR(base), IE_RIDE); /* make IE idle */
     outw(IE_GP(base), org);
     if ( (inb(EDLC_XMT(base))&(EDLC_JAM|EDLC_16)) == 0 ) return;
```

```
     outb(IE_CSR(base), IE_RIDE|mode);}

/* returns 0 on failure, 1 on success */
xmt_done(base, size, stall) short base, size, stall; {
     retry: switch(xmt_wait(base, size, stall)) {
          case 1: error("Transmit timed out"); iereset(base);
          case 0: return(1);
          case 2:
              error("Jam");
              retransmit(base, IE_XMTEDLC, size);
              sense_key();
              goto retry;
          case 3:   error("underflow on transmit"); iereset(base); break;
          case 4:   error("idle not set after xmt"); iereset(base); break;
          case 5:   iereset(base);
               error("excessive collisions");
               break;
          default: error("xmt_done: bad argument");}
     return(0);}

/* low level receive routines */
rcv_start(base, mode) short base; char mode; {
     outb(EDLC_RCV(base), EDLC_NONE);
     outb(IE_CSR(base), IE_RIDE);
     outw(IE_RP(base), 0);
     inb(EDLC_RCV(base));      /* he'll discard until we read the status */
     outb(IE_CSR(base), IE_RIDE|IE_RCVEDLC);
     outb(EDLC_RCV(base), mode|EDLC_GOOD);}

rcv_wait(base, stall) short base; int stall; {
     char status; int i = stall;
     do {
          if (inb(IE_CSR(base))&IE_RCVBSY) continue;
          status = inb(EDLC_RCV(base))&(EDLC_STALE|EDLC_RMASK);
          if ( (status&(EDLC_ANY|EDLC_RERROR)) != 0 ) return(status);}
     while(i-->=0);
     return(0);}          /* timed out */

rcv_chk(status) char status; {
     if (status&EDLC_FCS) error("FCS error");
     if (status&EDLC_DRIBBLE) error("dribble error");
     if (status&EDLC_OVER) error("overflow on receive");
     if (status&EDLC_SHORT) error("size");}

rcv_done(base, stall) short base; int stall; {
     char status;
     if ((status = rcv_wait(base, stall)) == 0) return(0);
     if (status < 0) {error("not fresh status"); return(-1);}
     outb(IE_CSR(base), IE_RIDE|IE_SYSBFR);  /* give buffer to processor */
     outb(EDLC_RCV(base), EDLC_NONE);   /* shut down the EDLC */
     rcv_chk(status);
     return(status&0xff);}     /* guaranteed to be non-zero at this point */

getaddr(base, cp) short base; char *cp; {int i;
```

```
     for(i=0; i<6; i++) {
          outw(IE_GP(base), i);
          *cp++= inp(IE_SAPROM(base));}}

setaddr(base, cp) short base; char *cp; {int i;
     for(i=0; i<=5; i++) outb(EDLC_ADDR(base)+i, cp[i]);}

/* fill packet with constant pattern */
fill_pkt(base, size, pat) short base, size, pat; {
     int i; char pathi = pat>>8;
     /* Watch out!  This routine knows that a short is two bytes. */
     outb(IE_CSR(base), IE_RIDE|IE_SYSBFR);
     size= (size+1) & ~1;      /* align packet on word boundary */
     outw(IE_GP(base), BFRSIZ-size);
     for(i=size>>1; i>0; i--) {
          outb(IE_BFR(base), pat); outb(IE_BFR(base), pathi);}}

xmt_pkt(base, size, stall) short base; int size, stall; {
     xmt_start(base, size); xmt_done(base, size, stall);}

rcv_pkt(base, rcv_mode) {
     int status, stallcon = 0x400;
     rcv_start(base, rcv_mode);
     while((status = rcv_done(base, stallcon))==0) sense_key();
     return(status);}

main() {
     char myaddr[6]; int ie = IE(0), size, i;

     /* one time only initialization */
     iereset(ie);          /* leaves buffer switched to system bus */
     getaddr(ie, myaddr);      /* read station address from PROM */
     setaddr(ie, myaddr);      /* set the station address */
     printf("3Com IE Programming Example Version 1.00);
     printf("My station address is ");
     for (i=0; i<6; i++) printf("%02x ", myaddr[i]&0xff);
     outb(EDLC_RCV(ie), EDLC_ALL|EDLC_GOOD);

     fill_pkt(ie, 1000, 0x5555);      /* fill packet with constant pattern */
     xmt_pkt(ie, 1000, 1000); /* transmit packet of 1000 bytes */

     /* receive those packets */
     printf("Start receive loop0);
     while (1)
          if (rcv_pkt(ie, EDLC_ALL|EDLC_GOOD) > 0) {
               size = inw(IE_RP(ie));    /* that's the size in bytes */
               printf("%d ", size);}
          else iereset(ie);}
```

5. Setting the Jumpers

The factory settings on the IE work with software supplied by 3Com. Only extraordinary circumstances or use with non-3Com software would require alteration of the factory settings.

The IE contains five sets of jumpers. Their functions and factory settings are summarized in the table below. The BNC/DIX jumper controls which of the two connectors on the backplate supports the Ethernet. The BNC is the silver cylindrical connector for a coaxial cable; the DIX connector is the fifteen pin connector. The DMA1/DMA3 jumpers select the DMA channel used by IE. The DMA function requires a pair of jumpers; be sure that the settings of the two jumpers agree. The INT3/INT5 jumper selects the interrupt channel used by the IE. IO<9-4> select the base address for the IE registers; the factory setting is 300, hex. PROM<19-12> select the base address of 4Kx8 PROM. The factory address setting is ec000, hex; however, the PROM is disabled.

BE CAREFUL when changing the jumpers. If the jumpers are improperly installed, it is possible to short together +5V and GND.

Function	Legend	Factory Setting
DIX/BNC	SW1	BNC
DMA1/DMA3	JP1	DMA1
	JP2	DMA1
INT3/INT5	JP3	INT5
IO9		1, not selectable
IO8	JP4	1
IO7	JP5	0
IO6	JP6	0
IO5	JP7	0
IO4	JP8	0
PROM19		1, not selectable
PROM18	JP9	1
PROM17	JP10	1
PROM16		0, not selectable
PROM15		1, not selectable
PROM14	JP11	1
PROM13	JP12	0
PROM12	JP13	0
PROM ENABLE	JP14	disable

6. Ethernet Interface

The IE provides two options for connection to an Ethernet, selectable by a jumper. The first is the standard, DA-15 DIX outlet, which uses fully compatible signalling. This outlet attaches to a standard transceiver cable, which in turn is connected to any Ethernet transceiver.

This would presumably be the usage when an Ethernet was pre-installed with "thick" Ethernet cable.

The other Ethernet interface uses the onboard transceiver and is designed to be used with, "thin", low-cost (50 ohm) RG-58A/U coax. The integral transceiver is attached to the Thin Ethernet cable via a single BNC connector on the box, to be mated with a BNC "T" pre-installed on the RG-58 coax. The station can be coupled and uncoupled without affecting network operation. The integral transceiver provides complete electrical isolation.

The RG-58 Ethernet is electrically compatible with the yellow Ethernet coax. In fact, the RG-58 Ethernet can be attached to a yellow Ethernet by simply coupling them with an N-series/BNC adapter, and IEs can communicate with any other station on the RG-58 or yellow coax. One drawback of the RG-58 Ethernet is that the distance limitation is more severe: approximately 300 meters of an RG-58-only segment.

7. Known Bugs Found on Early Production IEs

Receiving a runt packet can lock up the controller while in receive mode regardless of the setting of Detect Short Frames, bit three of the Receive Command Register. Reading the Receive Status Register after reception of a runt permits reception of further packets. This is not a problem for software using polled IO; such software can read the Receive Status Register in the same loop that checks the Auxilliary Status register for Receive Busy. Interrupt driven software must set Detect Short Frames to insure that runts generate interrupts; interrupt level software can then read the status register in order to receive subsequent packets.

It is possible to get one false interrupt for each write to the Receive or Transmit Command Register. Software can distinguish false interrupts from true ones by examining Receive Busy and Transmit Busy, bits zero and seven respectively of the Auxilliary Status Register. Well written software would routinely check for these conditions before taking action to disturb the state of the controller.

Software running at interrupt level can easily prevent false interrupts by reading the status register immediately after writing the control register.

Software running at main program level cannot prevent false interrupts that way because an interrupt would occur before main program level could read the status register. Interrupt level would then be responsible for clearing the interrupt by reading the status registers.