Technical Description of the EXOS 302 High-Performance VME Bus Ethernet Adapter

(Rev. 2 PCB and later)

Don Cohrac

Excelan, Inc.

[illegible]

1. Introduction

This document describes the EXOS 302 network processor board. This is a preliminary document which describes what will be included in the initial design.

2. General Description (Hardware Architecture)

The EXOS 302 is a microprocessor-controlled adapter which provides an Ethernet connection for computers using the VME bus. Architecturally it is compatible with the NX300 network software.

The EXOS 302 operates as a bus master, capable of reading and writing the host memory. Coordination between the host and the EXOS 302 is accomplished by the establishment of control blocks in host memory, the passing of values via special registers, and interrupts. The EXOS 302 operates as a slave to a bus master reading or writing its control registers.

The architecture is represented in Figure 1. Its major features are :

- o INTEL 80286 Microprocessor, 8MHz
- o INTEL 82586 Ethernet Controller
  - - IEEE 802.3 Compliance 10 Base 5
  - - ETHERNET V1 and V2 Compatibility
  - - D-type 15-pin transceiver connector
- o 512KB or 1MB Local Memory
- o 32KB or 64KB Local PROM for 80286 firmware
- o INTEL 8259A Interrupt Controller
- o Dual UART for RS232 port and timer
- o Host bus interface :
  - - 8- or 16-bit data
  - - 24- or 32-bit address
  - - Daisy-chained bus request/grant
  - - Daisy-chain interrupt structure
- o EXOS 202 compatibility mode (24-bit address)

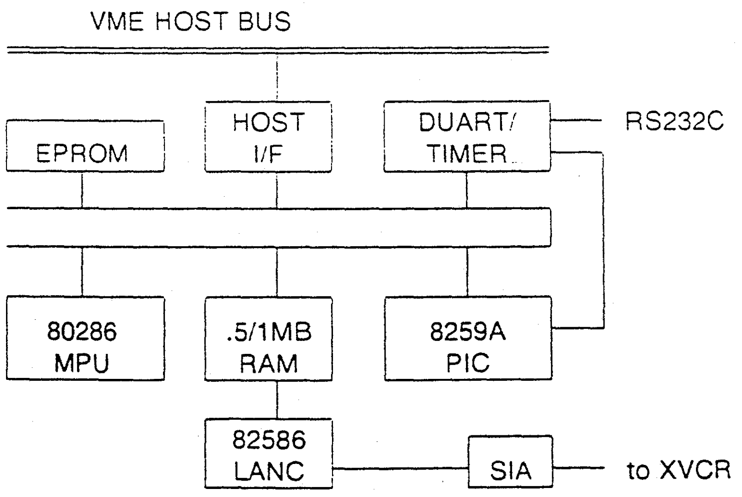

Figure 1. EXOS 302 Block Diagram

3. Functional Descriptiom

3.1. CPU

The processing element of the EXOS 302 is an Intel 80286 microprocessor. operating at an 8 MHz clock rate. It is supported by the 82284 clock chip, 82288 bus controller, and 8259A interrupt controller.

3.2. Memory Address Space

In real mode the 80286 has 1Mb of memory space, which is allocated as shown in Figure 2. The local memory appears as the lowest 512KB in the memory space. The host memory is mapped into the local space, and is limited to a relocatable 128KB. All local I/O devices reside in the 80286 I/O space (see section 4.1).

The 82586 can access only the dual-ported local memory; it can access the complete 512KB. It cannot access host memory.

In protected mode the 80286 can access beyond 1MB. An additional 512KB is provided at the beginning of the second megabyte. The 82586 can also access the additional 512KB.

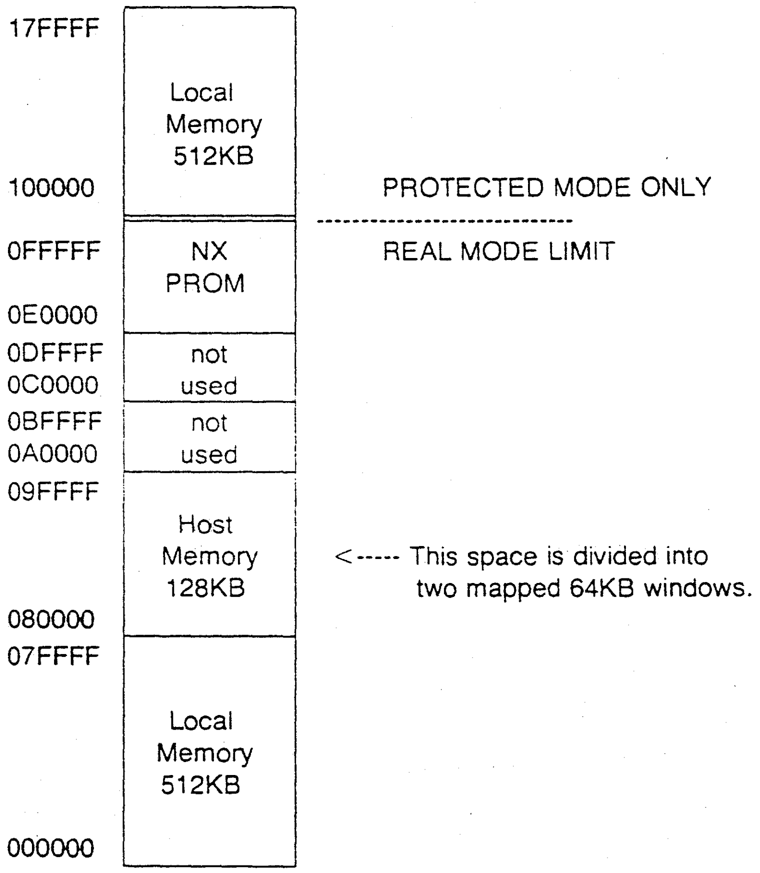

FIGURE 2. Memory Map

3.3. Local Memory

Local memory is two blocks of 512KB of dynamic RAM (total 1MB). The memory is 16 bits wide, and is dual-ported to the 80286 and the 82586. It is not accessible from the host bus. The memory has a minimum 1 wait state and maximum 6 wait states for the 82586, with an average of 3. It has a minimum of zero wait states and a maximum of 7 for the 80286, with an average of 1. Wait states above the minimum are caused by memory or refresh cycles-in-progress. CAS-before-RAS refresh is used.

The upper 512KB is accessible only when the 80286 is operating in the protected mode and a control bit is set in the EXOS Status Port (see section 4.1.11).

3.4. EPROM

Two sockets are provided for up to 64KB of EPROM for the 80286 firmware. EPROMs of 16K, 32K or 64K bytes can be used (see section 5.2.2).

3.5. Network Controller

The network controller performs the Ethernet control functions with minimum aid from the 80286. It consists of an Intel 82586 controller and an 8023A Ethernet Serial Interface chip.

The 80286 and the 82586 communicate via the local memory and the Channel Attention and network interrupt signals. Essentially the 80286 provides buffer descriptors for receiving and transmitting packets in the local memory, and generates the Channel Attention. Then the 82586 transmits or receives an Ethernet packet between the network and the local memory, and generates an interrupt to the 80286 when the operation is complete. Synchronization between the 80286 and the 82586 is maintained by means of the Channel Attention and interrupt. Neither the 80286 nor the 82586 can be locked out from accessing memory, so care is required in the implementation of semaphores. The 82586 documentation from INTEL describes the network interface and the 80286/82586 protocol in detail.

The 82586 reset signal is a latched bit, and is true upon reset, or by control of the 80286 via the CPU Control Port. The 82586 should be initialized before enabling the upper memory. Initialization data for the 82586 must be set up at memory location 007FFF6 (system configuration pointer) prior to the first assertion of Channel Attention. If the 80586 attempts to fetch the configuration pointer while the upper memory is enabled, it will fetch from location 017FFF6 (address bit 20 is enabled, and address bit 19 is ignored because of the discontinuity in memory between 512K and 1M).

The hardware provides two loopback features for confidence test capability. A bit in the CPU Control Port enables loopback through the transceiver interface chip. The 82586 also has facilities for internal loopback and diagnostics (see Intel documentation).

3.6. DUART/Timer

An RS232C serial port for debug purposes and a periodic interrupt timer are provided by a single LSI chip, Signetics SCN2681. See section 4.1.1 for details of the implementation.

3.7. Host Interface

3.7.1. VME Bus Conformity

EXOS 302 VME bus conformity is :

```
       A24 MASTER, D(EO), D16;  A24 SLAVE, D(O)
   OR  A32 MASTER, D(EO), D16;  A24 SLAVE, D(O)
   OR  A32 MASTER, D(EO), D16;  A32 SLAVE, D(O)
```

depending upon jumper configuration.

The EXOS 302 as a slave responds to two sets of address modifier codes. The slave bus size jumper defines which set of codes is valid. These codes are implemented with a PAL, so can easily be modified.

A24 SLAVE : address modifier codes (hex) 39, 3A, 3D, 3E.
A32 SLAVE : address modifier codes (hex) 09, 0A, 0D, 0E.

3.7.2. Address Generation

The EXOS 302 acts as a bus master to transfer data between its memory and the system memory over the host bus. The EXOS 302 provides 32 address lines to the host, for a total addressable memory space of 4 Gigabytes. However, only 128KB is accessible at a time. The lower sixteen bits from the 80286 are concatenated with sixteen bits from a mapping register to form the full 32-bit address. Address line 16 from the 80286 selects which of two sixteen-bit registers is used (see Figure 3). This provides two 64KB windows, each of which can be placed on any 64KB boundary in the host's address space. The two windows appear as locations 080000-08FFFF and 090000-09FFFF to the 80286 (see Figure 2).

In the 32-bit mode the six address modifier bits, which must be driven by a bus master, are derived from an address modifier register which is written by the host (see section 4.2).

The EXOS 302 can be jumpered to operate as a 24-bit master, which makes it compatible with the EXOS 202 host driver and address modifier scheme. The difference from the 32-bit mode, besides the number of address bits, is the source of the six address modifier bits. In 202 mode the address modifiers are derived from the low six bits of the upper byte of the mapping register (see Figure 3).

Note that the EXOS 302 is HARDWARE-configured as either a 32- or 24-bit master. If configured for 32-bit. the address modifier is always driven by the addesss modifier register. 24-bit memories can be addressed by changing the address modifier register to one valid for 24-bit devices. The upper byte of the map register as the address modifier source in this case, since the map register is driving sixteen bits of address.

In 24-bit mode the address modifier from the map register is also driven onto the upper address byte (bits 31-24). According to the VME specification, if the address modifier indicates a 24-bit address, devices must not decode the upper byte.

The EXOS 302 can do eight- or sixteen-bit data operations. Bytes are automatically swapped by the hardware (high byte with low, low with high) since it is assumed that any memory device will present data to the bus in 68000 format.

```
         enabled if host is     286 data    286 addresses
         ------------------     --------    -------------
           32bits  24bits        D(0:15)     A16  A(1:15)
              |     |               |         |      |
              |     |        .------+-------.`|      |
              |     |        |MAP REGISTER +-'       |
              |     |        |  16 X 2     |         |
   .-----.    |     |        `-+---------+-'         |
   |     |    |     |          |8        |           |
   |   .-+----+-.   |   .------+         |           |
   |   | ADDRESS|   |   |6     |8        |8          |16
   |   |MODIFIER| .-+---+.  .--+---.  .--+---.    .--+---.
   |   |REGISTER| |BUFFER|  |BUFFER|  |BUFFER|    |BUFFER|
   |   `----+---' `---+--'  `--+---'  `--+---'    `--+---'
   |        `----+----'        |         |           |
  /|\           \|/           \|/       \|/         \|/
   |             |             |         |           |
VD(0:5)      VAM(0:5)      VA(31:24) VA(23:16)   VA(15:1)
```

FIGURE 3. EXOS 302 Bus Master Address Generation

3.7.3. Slave Ports

Masters on the host bus can access three I/O ports on the EXOS 302. These slave ports are used for resetting the board, setting the address modifier register, setting the interrupt vector, and reading or writing program-defined status. These ports are compatible with the EXOS 202 ports, with the addition of the address modifier register.

The EXOS 302 is able to read and write its own ports, except for the interrupt vector, which is accessible only as an interrupt vector. The address modifier register can be written by the EXOS 302 under certain circumstances. Before the register is written, its contents are undefined. If the EXOS 302 does a host bus write cycle, it will likely assert a garbage address modifier. This will prevent the EXOS 302 slave machine from responding, and the bus will hang (or timeout).

The slave bus size can be configured separately from the master size. This allows both 24-bit, both 32-bit, or 24-bit slave with 32-bit master (slave 32 and master 24 is prohibited by the logic). This is accomplished by two jumpers (see section 5.2).

3.7.4. Bus Errors

The bus error signal is not generated by the EXOS 302, since access is only to registers. The only error that can occur is an incorrect data alignment access by the host (double-byte or quad-byte). On such an access the EXOS 302 does not generate transfer acknowledge (DTACK), and a system bus timeout should occur. The EXOS 302 does monitor the bus error when a master, and an error causes the NMI to the 80286. The NMI is enabled via the CPU Control Port.

Signals ACFAIL and SYSFAIL are monitored, and while either is asserted the EXOS 302 bus request is inhibited. If the 80286 requests the bus during this time, it is held in wait.

3.7.5. Burst Mode

The EXOS 302 has a burst mode, which is enabled by a jumper (see section 5.3). It holds the bus a maximum of 15 microseconds, unless the 80286 stops requesting the bus sooner. This allows about 20 cycles. Since the bus will be released regularly, the bus clear signal is not monitored. Note that this is not a VME BLOCK operation, wherein the slave incrèments the address itself after the first transfer. The EXOS 302 generates an address every cycle. The bus busy line is simply held true to avoid arbitration for each cycle.

Normally, burst mode will not be effective. However, under certain conditions burst mode may increase performance. These are, either singly or in combination : if the bus arbiter has a request-to-grant latency greater than 200 nanoseconds; if some masters in the system do not provide early release of busy, thereby preventing pipelined arbitration; if the EXOS 302 is not at the beginning of the grant daisy-chain.

3.7.6. Interrupts

The EXOS 302 implements a daisy-chained interrupt structure. All bus interrupts (1-7) are available. The interrupt level is selected by a jumper. Three other jumpers are used to decode the interrupt acknowledge. See section 5.2 for jumper configurations.

The 80286 can issue a host bus interrupt by writing to an internal port (see section 4.1). The interrupt request is cleared by the hardware during interrupt acknowledge sequence (ROAK).

An interrupt to the EXOS 302 from the host is also available. This interrupt is set when one byte of data is written to the Comm register in the EXOS 302. See section 4.2.

3.8. Indicators (LED)

The EXOS 302 has three red LEDs to indicate operational status; one for network transmit activity, one for host bus activity, and one diagnostic LED which is connected to the CPU Control Register. Figure 4 shows the position of these LEDs on the board.

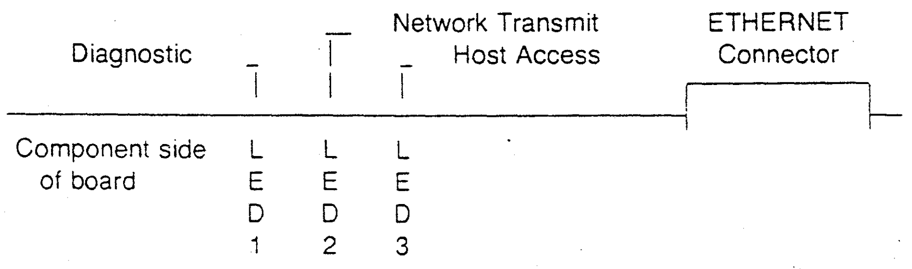

Figure 4. LED Positions

4. Programming

This section explains the I/O ports which are used to control the operation of the EXOS 302. It contains two subsections: one for internal control, and one for host bus.

4.1. CPU Internal I/O

The I/O map is shown below. An I/O device uses some or all the I/O addresses in the range. When less than all the space is used, data replicates. Since incomplete decoding is used, the entire I/O map from 00-FF repeats to 7FFF.

Note that all the I/O devices are 8-bit except for the CPU Configuration/ Status Port. This means that all I/O addresses are even (except for the upper byte of the Configuration/Status Port). For the sake of simplicity, all I/O devices will be given one wait state.

A caveat using I/O: the MOS devices (PIC and DUARTS) require recovery time between I/O commands. Because of the speed of the 80286, NOP's will be required when accessing these devices. The details are described in the appropriate sections of this document.

Base Address	Type	Width in bits	Function
00	Rd/Wr	8	DUART (2681)
20-80			reserved
88	Write	-	Channel Attention
90	Rd/Wr	8	CPU Control Register
98	Rd/Wr	16	CPU Status Register
A0	Rd/Wr	8	Interrupt Controller (8259A)
B0	Write	-	Schedule Interrupt (software)
B8	Write	-	Interrupt Host
C0	Read	8	Ethernet Address PROM
E0	Write	16	Address Mapping Registers
E8	Write	-	Local Reset of EXOS 302
E8	Read	8	Host data register and interrupt reset
F0	Write	16	EXOS 302 Status
100			00-FF Repeats to FFFF

FIGURE 3. I/O Map

4.1.1. DUART (Serial port/timer) (Read/Write at 00h)

The serial port and timer are implemented using a Signetics SCN2681 Dual Universal Asynchronous Receiver/Transmitter (!) chip. The chart below shows the connection of the chip to the RS232C port and the 8259A. Specific programming information for the chip can be found in the Signetics Microprocessor Data Manual (1986). The chip is operated from a 3.6864 MHz crystal. The interrupt request is connected to INT2 of the 8259A. Note that the chip is on the upper byte of the data bus. Therefore its ports are every other odd address. One NOP is required between successive I/O operations to the DUART to satisfy command recovery time.

2681 Pin	Function
RXA	Receive Data
TXA	Transmit Data
IP0	Clear to Send
IP4	Data Terminal Ready
OP0	Request to Send
OP2	Data Set Ready
OP3	Counter output to INT2

4.1.2. Channel Attention (Write at 88h)

A write to this port (data is irrelevant) generates a pulse to the Channel Attention input of the 82586.

4.1.3. CPU Control Port (Read/Write at 90h)

The CPU Control Port is a latching register used by the 80286 for internal control of the EXOS 302. All bits are set to zero at power up or reset by the host. The port can be written and read at I/O address 90, and is defined as follows:

Bit	Value	Function
0	0	Reset 82586 network controller
	1	Enable 82586 network controller
1	0	Enable loopback mode of 8023 serial I/F
	1	Enable normal mode of 8023 serial I/F
2	0	Disable NMI (bus parity error/timeout)
	1	Enable NMI
3	0	Light diagnostic LED
	1	Extinguish diagnostic LED
4		not used
5		not used
6		not used
7		Diagnostic bus control (see section 5.2.3)

4.1.4. CPU Configuration/Status Port (Read at 98h)

A 16-bit I/O port at address 98 is assigned to configuration jumpers and other status bits. These bits are defined as follows:

Bit	Source	Function
0	sense	Xcvr +12V fuse (0=blown; 1=ok)
1	sense	not used
2	sense	not used
3	sense	Bus error (0=parity; 1=spurious)
4	sense	Mem size (1=512KB; 0=1MB)
5	sense	not used
6	sense	not used
7	sense	CPU Clock (1=8MHz; 0=n/a)
8	J2	Disable CRS check (=0)
9	J3	Ethernet/no SQE test (=0)
10	J4	Boot (0=from net; 1= wait for host)
11	J5	reserved
12	J6	reserved
13	J7	reserved
14	J8	NX Console (=0)
15	J9	Debugger (=0)

All jumper inputs are pulled up (=1). An installed jumper is read as a zero (=0).

4.1.5. Interrupt Controller (PIC) (Read/Write at A0h)

The 8259A Interrupt Controller provides prioritized interrupts to the 80286. Refer to Intel literature for operation and description of its I/O registers. Note that due to timing constraints (Intel's - not ours), one no-op is required between successive identical commands, and two no-ops are required between successive dissimilar commands. This is required to satisfy the chip's command recovery time. Since the 8259A is a byte device, it uses even addresses A0h & A2h. Interrupt sources are listed below.

Level	Source
INT0	reserved
INT1	Host Data Register/Interrupt
INT2	2681 Output OP3 (Timer)
INT3	82586
INT4	2681 Interrupt (SIO)
INT5	
INT6	Scheduling Int
INT7	

4.1.6. Scheduling Interrupt (Write at B0h)

An access to this port (data is irrelevant) generates an interrupt on INT6 of the 8259A.

4.1.7. EXOS 302 Interrupt (Write at B8h)

A write to this port (data is irrelevant) generates the EXOS 302 interrupt to the host. It also sets a status bit which appears in the EXOS 302 Status port for the host. The interrupt is cleared by a reset or interrupt acknowledge. The status bit is cleared by reset or by command from the host (see section 4.2).

4.1.8. Ethernet Address PROM (Read At C0-DF)

The address PROM contains the Ethernet address and other configuration information. Sixteen bytes are available for use (even addresses only). A possible format is illustrated below:

Address	Contents
C0	PROM rev level
C2	Byte 5 of address
C4	Byte 4 of address
C6	Byte 3 of address
C8	Byte 2 of address
CA	Byte 1 of address
CC	Byte 0 of address
CE	Product type
D0	Hardware rev level
D2-DD	reserved
DE	Checksum

The product type for the EXOS 302 is 12 (decimal). The checksum is formed by summing all the bytes modulo 256 and exclusive or'ing with 55. This procedure ensures that a PROM with all zeros or all ones will not have a correct checksum.

4.1.9. Extended Address Map Registers (Write at E0h)

Accesses to the host use 24 or 32 addresses, of which the lower 16 come from the 80286. The upper 16 are provided from one of two 16-bit mapping registers.

When host bus accesses are performed (addresses 8000-9FFF), address line 16 from the 80286 selects which register is concatenated with the lower 16 addresses. The mapping registers are loaded by writing data at I/O locations E0 and E2. The write data must be 16 bits wide. The lower byte furnishes host address bits 16-23. The upper byte furnishes bits 24-31 if in 32-bit-address mode. If jumpered for EXOS 202 compatibility, the upper byte furnishes address modifier bits 0-5 (register bit 8= address modifier bit 0).

Map Register	Maps CPU Addresses	Master Mode	Map Register Bits 15-8	Map Register Bits 7-0
E0	8000-8FFF	32-bit	A31-24	A23-16
E2	9000-9FFF	EXOS202	AM5-0	A23-16

4.1.10. Host Data Register and Interrupt Reset (Read at E8h)

A read of this port acquires the byte written into Port B when the host set the interrupt to the EXOS 302. Reading this port clears the interrupt to the 8259A, and clears status bit 3 of Port B.

4.1.11. Local Reset of EXOS 302 (Write at E8h)

A write to this port (data is irrelevant) generates the board reset. This is equivalent to a host reset, and resets the entire board. The 80286 returns to real mode.

4.1.12. EXOS 302 Status Bytes 0 and 1 (Write at F0h, Read as Ports A and B)

A 16-bit write-only register at location F0h is available to provide status information to the host Ports A and B. All bits are software-definable. (Current definition of the status bits is explained in the Host Interface section.) However, bits 1 and 3 of Port B do not come from this register. They are derived directly from the hardware, and indicate the state of the EXOS 302 interrupt and the host interrupt, respectively. Reset clears bits 0-7; bits 8-15 are not cleared.

Bit 1 of the write-only register is used internally to enable the upper 512KB RAM when the 80286 is operated in the protected mode. This inhibits access to the upper RAM. When the bit is set, access is enabled. The bit should be set AFTER the 80286 is in protected mode, because the extended address bits from the 80286 in real mode are not in a defined state, and may cause faulty memory operation.

Bit	Destination	Definition
0	Port B	0=diagnostic failed; 1=passed
1	Memory Control	0=disable upper 512K; 1=enable
2	Port B	
3	no connect	
4	Port B	none
5	Port B	none
6	Port B	0=loopback passed; 1= failed
7	Port B	none
8-15	Port A	none

Bit 3 is not connected to anything internally. Bits 8-15 are all wired directly to Port A. This register must always be written as a word register; byte operations will trash the other byte.

4.2. Host Slave Ports

Three ports exist on the EXOS 302 in the host memory-mapped I/O space. Two are for status and control information to/from the EXOS 302. The third is for the host to write the 6-bit address modifier for 32-bit address mode.

There are individual compare jumpers for address bits 31-16, and two jumpers to define one of four combinations for bits 15-7. These four combinations are programmed into a PAL, and currently are compatible with the EXOS 202 rev F. The slave address configuration is shown below. (This is a program representation - bit 0 does not appear on the VMEbus.) Bits 31-24 are not decoded when in 24-bit slave mode. Bits 6-3 are not decoded, leaving a 128-byte hole above the base address. These ports must be accessed as byte values.

```
31        24   23        16   15        8   7        0
  aaaa aaaa  |  aaaa aaaa  |  bbbb bbbb  |   b--- -ppp
```

Port	Base Offset	Type	Function
A	−1	Read	Read Status Byte 1 (extended status)
A	+1	Write	EXOS 302 Reset
B	−3	Read	Read Status Byte 0
B	+3	Write	Host Interrupt/Data Write
C	−5	Write	Write Address Modifier Register

Write Port A Generates a hardware reset of the EXOS 302.

Read Port A Reads the EXOS 302 Status Byte 1 (extended status). All bits are register bits, and can be set by the EXOS 302 firmware. Contents are indeterminate after reset. See section 4.1.

Write Port B Host Interrupt/Data - Interrupts the 80286 via INT1 of the 8259A and writes a byte into the Host Data Register. This also sets the host interrupt status bit, which is available to the host in the EXOS 302 Status port. The bit is cleared when the 80286 reads the Host Interrupt/Data Port. This mechanism provides for handshaking and parameter passing between the EXOS 302 and the host.

Read Port B Reads the EXOS 302 Status Byte 0. All the bits are cleared by reset, except bit 3, which is set. All bits are register bits which can be set by the EXOS 302 firmware, except bits 1 and 3, which are interrupt status bits (see section 4.1).

Bit 0 Set/reset by EXOS 302 firmware. This bit has previously been defined as Status of the board. When reset, indicates the on-board diagnostic has failed or is not completed yet.

Bit 1 Host Interrupt Status - this bit is set by the hardware when the 80286 issues an interrupt to the host. It is cleared by the hardware during the interrupt acknowledge sequence.

Bit 3 Local Interrupt Status - is set when the host writes into Port B, and is cleared when the 80286 reads the Host Data Port. On power reset the state of this bit is undefined. Therefore, the 80286 should read the Host Data Port before enabling interrupts.

Bit 6 Set/reset by EXOS 302 firmware. This bit has previously been defined as a qualifer of bit 0. When set, indicates a failure of net loopback.

Write Port C A byte is written into the address modifier register in the EXOS 302. Only the low six bits are valid. This register drives the six address modifier bits during a bus cycle if the master size is jumpered for 32 bits.

5. Board Configuration

5.1. Memory Option

An additional 512KB of memory may be installed in sockets U16-U19. The memory size jumper in the CPU Status Register must agree with the installed size. The chips must be 256K X 4 DRAMS, 120ns or faster, with CAS-before-RAS refresh capability (TOSHIBA 514256 or equivalent).

5.2. Jumpers

5.2.1. CPU Status Register Jumpers

All jumper inputs are pulled up (1). An installed jumper is read as zero (0). Note that these are only sense jumpers, and have no affect on the hardware, except J3 and J10.

Bit	Jumper	Function
4	J10	Mem size (1=512K; 0=1MB)
5	J11	reserved (=1)
6	J12	reserved (=1)
7	J13	286 Clock (1=8MHz; 0=n/a)
8	J2	Disable CRS check (=0)
9	J3	V1 ethernet/no SQE test (=0)
10	J4	Net Boot (=0)
11	J5	reserved (=1)
12	J6	reserved (=1)
13	J7	reserved (=1)
14	J8	NX Console (=0)
15	J9	Debugger (=0)

5.2.2. PROM Size

Two jumpers define the size of the PROMs used for the 80286 firmware.

PROM Type	J16	J15
27128	1-2	1-2
27256	1-2	2-3
27512	2-3	2-3

5.2.3. Burst Mode Enable

This jumper enables or disables the burst mode to the host bus.

Burst Function	J25
Enabled	1-2
Disabled	2-3

5.2.4. Address Configuration Jumpers

There are two sets of address jumpers to configure : one for selecting the host bus address width (24 or 32 bits) for slave and/or master mode, and one for slave mode decode.

5.2.4.1. Host Address Bus Width

The host address bus width determines how many bits are decoded on a slave cycle, and the source of the address modifier on a master cycle. Master and slave mode can use different widths. Note that the master mode cannot be smaller than the slave. However, if the address modifier register contains a 24-bit modifier, then only 24-bit slaves can respond.

Master Width (bits)	Slave Width (bits)	Master Jumper J44	Slave Jumper J46
24	24	ON	ON
32	24	OFF	ON
32	32	ON	OFF
32	32	OFF	OFF

5.2.4.2. Slave Address Compare

There are individual compare jumpers for address bits 31-16, and two jumpers to define the base within 64KB (bits 15-7). The two jumpers select one of four bases which are programmed into the decode PAL.

The corresponding addresses/jumpers are shown in the two charts below. For the address compare jumpers, an installed jumper is a compare to zero. The four base addresses are compatible with the EXOS202 (rev F). Bits 6-3 are not decoded, leaving a 128-byte hole above the base address.

	Host Address Bits					
A	31 30 29 28 27 26 25 24	23 22 21 20 19 18 17 16	15 - 8	7 - 0		
J	26 27 28 29 30 31 32 33	34 35 36 37 38 39 40 41			42	43
	Slave Address Compare Jumpers					
			00	00	OFF	OFF
			7F	80	OFF	ON
			80	00	ON	OFF
			FF	80	ON	ON

5.2.5. Interrupt Level and Level Enable

The request jumper and the acknowledge jumpers MUST match for interrupt operation. An installed jumper is indicated by a 1.

IRQ	Request	Acknowledge Jumpers		
Level	Jumper	J54	J55	J56
IRQ1	J47	0	0	1
IRQ2	J48	0	1	0
IRQ3	J49	0	1	1
IRQ4	J50	1	0	0
IRQ5	J51	1	0	1
IRQ6	J52	1	1	0
IRQ7	J53	1	1	1

5.2.6. Bus Request/Grant In/Grant Out

Request and grant jumpers must be on the same level. One Request jumper is used. One Grant In and one Grant Out are jumpered to match the request level. The other three Grant Out lines must be jumpered to daisy-chain the unused Grant In lines back to their respective Grant Out lines.

	Request	Grant In/Out Jumpers							
Level	Jumper	J68	J67	J66	J65	J64	J63	J62	J61
0	J60	ON	ON	J65-1	J66-1	J63-1	J64-1	J61-1	J62-1
1	J59	J67-1	J68-1	ON	ON	J63-1	J64-1	J62-1	J61-1
2	J58	J67-1	J68-1	J65-1	J66-1	ON	ON	J61-1	J62-1
3	J57	J67-1	J68-1	J65-1	J66-1	J63-1	J64-1	ON	ON

5.2.7. Host Bus Ethernet Connection

The Ethernet signals connected to the 15-pin D-type connector can be jumpered to the host bus on P2. All jumpers must be installed if used. The −12V is fused (VE−12V).

Signal	Jumper	Host Pin
VGND	J24	P2C-15
VE−12V	J17	P2C-16
VTRMT+	J18	P2A-13
VTRMT-	J19	P2A-14
VRECV+	J20	P2A-15
VRECV-	J21	P2A-16
VCLSN+	J22	P2C-13
VCLSN-	J23	P2C-14

5.2.8. Dynamic Address Bus Control (Diagnostic only)

The diagnostic jumper J44 is to be used only in the factory with special test equipment which can short the jumper upon command. The jumper, when shorted, allows the firmware to force the board from 24-bit mode into 32-bit mode by setting the DIAG24 bit in the CPU Control Register. This is necessary for testing the address modifier register, since testing will involve using invalid modifiers, and an invalid modifier will prevent subsequent slave cycles. Testing is accomplished by 1) forcing 24-bit mode to use address modifier from the map register, 2) writing the address modifier register with a test value, 3) switching back to 32-bit mode to use the address modifier register, 4) doing a bus cycle, which is latched in the bus tester, and 5) reading the latched address modifier value from the bus tester. The master jumper is pulled up, and, if off, indicates 32-bit mode. The master will follow the slave if the Master jumper is on. This circuit is shown below. Note that the DIAG24 and SLAVE24 signals are low true, and MASTER32 is high true. The gate is a positive AND. Note that this process cannot force the board into 32-bit mode if the slave jumper is set for 24-bit mode.

The DIAG24 bit is 0 after reset. The CPU Control Register is a write/read register. However, the DIAG24 bit (7) is not read directly. The bit which is read is an AND of the DIAG24 bit and the DIAG jumper.

```
                          ^ Pullup
+------+                  |          +------+
| WRT  |  DIAG24-   J45   |          | READ |
| REG  +------------0 0---+--+-------+ REG  |
| b7   |                     |       |  b7  |
+------+                     |       +------+
                             |      ___
                       ^     |     |   \                   ^
                       |     |     |    \                  |
                       |     +-----|     \    SLAVE24-     | MASTER32
               J46     | BUS24-    | AND|-------------0 0--+----------
           +--0 0------+-----------|    /        J44
           |                       |___/
          GND
```

6. Connectors

6.1. VME Bus Connectors

The following table shows the EXOS 302 signal assignments for the VME host bus connectors P1 and P2. Signals not used by the EXOS 302 are designated n/u. Many of the pins on P2 are not dedicated to any bus signals.

VME BUS PIN ASSIGNMENTS						
PIN	P1A	P1B	P1C	P2A	P2B	P2C
1	VD00	VBSY-	VD08		+5V	
2	VD01	BCLR- n/u	VD09		GND	
3	VD02	ACFAIL-	VD10			
4	VD03	VBG0IN-	VD11		VA24	
5	VD04	VBG0OUT-	VD12		VA25	
6	VD05	VBG1IN-	VD13		VA26	
7	VD06	VBG1OUT-	VD14		VA27	
8	VD07	VBG2IN-	VD15		VA28	
9	GND	VBG2OUT-	GND		VA29	
10	SYSCLK n/u	VBG3IN-	SYSFAIL-		VA30	
11	GND	VBG3OUT-	VBERR-		VA31	
12	VDS1-	VBR0-	SYSRESET-		GND	
13	VDS0-	VBR1-	LWORD-	VTRMT+	+5V	VCLSN+
14	VWRITE-	VBR2-	VAM5	VTRMT-	D16 n/u	VCLSN-
15	GND	VBR3-	VA23	VRECV+	D17 n/u	VGND
16	VDTACK-	VAM0	VA22	VRECV-	D18 n/u	VE-12V
17	GND	VAM1	VA21		D19 n/u	
18	VAS	VAM2	VA20		D20 n/u	
19	GND	VAM3	VA19		D21 n/u	
20	IACK-	GND	VA18		D22 n/u	
21	IACKIN-	SERCLK n/u	VA17		D23 n/u	
22	IACKOUT-	SERDAT n/u	VA16		GND	
23	VAM5	GND	VA15		D24 n/u	
24	VA07	IRQ7-	VA14		D25 n/u	
25	VA06	IRQ6-	VA13		D26 n/u	
26	VA05	IRQ5-	VA12		D27 n/u	
27	VA04	IRQ4-	VA11		D28 n/u	
28	VA03	IRQ3-	VA10		D29 n/u	
29	VA02	IRQ2-	VA09		D30 n/u	
30	VA01	IRQ1-	VA08		D31 n/u	
31	-12v	+5V STBY n/u	+12V		GND	
32	+5v	+5V	+5V		+5V	

6.1.1. ETHERNET Connector

The ETHERNET connector P3 is a 15-pin D-type. The following table gives the pin assignments.

P3	
Pin	Signal
1	GND
2	CLSN+
3	TRMT+
4	GND
5	RCV+
6	GND
7	
8	GND
9	CLSN-
10	TRMT-
11	GND
12	RCV-
13	E+12V
14	GND
15	

6.1.2. SERIAL PORT HEADER

The following diagrams show the pin orientation and signal assignments for P4, the serial port header.

Top edge of board, IC side

Pin	25 23	...	3 1	L
Nos. >	26 24	...	4 2	E
				Ds

P4	
Pin	Signal
1	
2	
3	RXD
4	
5	TXD
6	
7	CTS
8	
9	RTS
10	
11	DTR
12	
13	GND
14	DSR
15	
16-26	not used

```
        PAGE    55,132

COMMENT '

$Header: c:/nx6/s_net/net586/RCS/net586_2.asm 1.3.1.1 91/03/07 10:21:10 karlt Exp Locker: karlt

$Project: NX6 $

$Creator: Steve Grau $

$Locker: karlt $

$Source: c:/nx6/s_net/net586/RCS/net586_2.asm $

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

(C) Copyright 1988 by Excelan Inc. All Rights Reserved

This software is furnished under contract and may be used and copied
only in accordance with the terms of such contract and with the
inclusion of the above copyright notice. This software or any other
copies thereof may not be provided or otherwise made available to
any other person. No title to and ownership of the software is hereby
transferred.
-----------------------------------------------------------------------------

$Abstract:

        NX6 Intel 82586 Ethernet Controller Driver.

        This module contains the transmitter state machines.
$


$Implementation Notes:

        The 82586 transmitter driver is implemented using four state machines
to manage the transmit queues and keep the 82586 running at maximum
throughput.

Transmitter Queues

The transmitter contains two queues.  The first queue is made up of a circular
list of transmit command blocks (CBLs) and a circular list of transmit buffer
descriptors (TBDs).  These structures (CBLs and TBDs) are described in
detail in the 82586 data sheet.  The two circular lists do not necessarily
contain the same number of elements.  Fragmentation of transmit buffers
requires several TBDs per CBL.  So the TBD list should be much longer than
the CBL list.  The actual sizes of these lists is configurable at assembly
time.  This queue will be referred to as the Transmit Ring.

The second queue is simply a linked list which is used to hold user requests
when there is a resource limitation in the first queue (out of CBLs or out
of TBDs, or both).  This queue will be referred to as the Wait Queue.

Transmitter State Machines

There are four transmitter state machines.  They are:

        Enqueue Transmit - (entry points qtx_????)
        ;       Handles queing user transmit requests to the appropriate
                transmit queue.  Also, starts the 82586 transmitting if
                it is not already doing so.

        Transmit Complete - (entry points cx_????)

                Runs on the command complete interrupt, CX bit in the
                82586 interrupt status.  Notifies, the user when each
                transmit completes and frees up the CBL and TBDs used
                by the transmit.

        Process Wait Queue - (entry points pwq_????)

                Transfers transmit requests from the wait queue to the
                transmit ring as ring resources become available.  Is
                run on the same interrupt as the transmit complete
                state machine immediately after it finishes.

        Transmitter Not Active - (entry points cna_????)

                Restarts the 82586 transmitter when it completes a chain
                of transmit requests.  All requests waiting in the transmit
                ring are forwarded to the 82586.  Gets invoked by the
                command not active (CNA) interrupt from the 82586.

In addition to the state machines there is also a transmit timeout interrupt
that runs of a clock interrupt.

$


$Log:   net586_2.asm $
Revision 1.3.1.1  91/03/07  10:21:10  karlt
temporary fix for APPLLO's transmit hang problem

Revision 1.3  89/02/22  09:04:49  steveg
Added new TPS status bit.

Revision 1.2  88/12/09  15:57:31  dauber
added support for concurrent link level

Revision 1.1  88/08/23  08:40:13  steveg
Initial revision


$EndLog$

'


include nx6.inc
include cfgxxx_x.inc
include cfg_x.inc
include intxxx_x.inc
include net586.inc

        .186

_TEXT           SEGMENT
```

```
EXTRN   net586_qtx_return:NEAR                  ; qtx state machine return pt.
EXTRN   net586_cna_return:NEAR                  ; cna state machine return pt.
EXTRN   net586_cx_return:NEAR                   ; cx state machine return pt.
EXTRN   net586_pwq_return:NEAR                  ; pwq state machine return pt.

public cx_state
public cna_state
public pwq_state
public qtx_state
public p_1st_586_cbl
public p_1st_rdy_cbl
public p_last_rdy_cbl
public p_1st_free_cbl
public p_1st_tx_wait
public p_last_tx_wait
public p_1st_free_tbd
public p_last_free_tbd
public last_tbs_off
public last_tbs_sel
public fragment_count

;****************************************************************************
; net586_tx_chain - return transmit buffer chain
;
; This function returns the TPS and TBS structures that have been queued
; to the driver.  The transmit is turned off.
;
;       Inputs: none
;
;       Outputs: STACK_USER_ES:STACK_SI points to TPS chain
;                STACK_SI = ffff if none returned
;                STACK_AX = 0
;
;       The TPS chain is linked by the QUEUE field.  The end of the chain
;       is marked by an offset of ffff.
;
;****************************************************************************

        PUBLIC  net586_tx_chain
net586_tx_chain PROC    NEAR

        pushf                                   ; protect
        cli

; Disable transmit timeout.

        mov     tx_timeout_enable,0             ; disable the timeout

; Abort the transmitter.

        mov     cx,0ffffh                       ; long timeout
txc_wait_cmd:
        cmp     scb.SCB_COMMAND,0               ; command clear?
        je      txc_cmd_clear                   ; yes
        loop    txc_wait_cmd

txc_cmd_clear:
        mov     scb.SCB_COMMAND,SCB_CUC_ABORT   ; abort the transmitter
        CHANNEL_ATTENTION                       ; bang on the 586
        mov     si,0ffffh                       ; assume no buffers
        mov     WORD PTR [bp+STACK_SI],si

; Check to see if there are any tps's on the tps wait queue.

        cmp     pwq_state,OFFSET pwq_wait       ; tps wait queue active?
        jne     txc_tx_wait_done

; Get the pointer to first waiting tps.

        mov     ax,p_1st_tx_wait                ; get the offset
        mov     [bp+STACK_SI],ax
        mov     ax,p_1st_tx_wait+2              ; get selector
        mov     [bp+STACK_USER_ES],ax

        mov     si,p_last_tx_wait               ; get pointer to last waiter
        mov     es,p_last_tx_wait+2

txc_tx_wait_done:

; Check to see if there are any tps's queued to the 82586.

        cmp     cna_state,OFFSET cna_idle       ; 586 transmitting?
        je      txc_tps_done                    ; no

; Free up the tbds.

        mov     bx,OFFSET tbd_list              ; point anywhere in list
        mov     p_last_free_tbd,bx              ; this one is last...
        mov     bx,[bx].TBD_LINK                ; ..next one is first
        mov     p_1st_free_tbd,bx

; Shuffle through the cbls linking tps's together.

        mov     bx,p_1st_586_cbl                ; get pointer to first cbl

txc_cbl_loop:
        cmp     bx,p_1st_free_cbl               ; all cbls done?
        je      txc_tps_done                    ; yes

        cmp     p_1st_free_cbl,OFFSET p_1st_free_cbl ; any free?
        jne     txc_some_free                   ; yes
        mov     p_1st_free_cbl,bx               ; make this the first free one

txc_some_free:

        cmp     si,0ffffh                       ; any on the return list?
        jne     txc_got_some                    ; yes

; The return chain is empty.  Attach the first element.

        les     si,DWORD PTR [bx].CBL_TPS_OFF   ; get pointer
        mov     [bp+STACK_SI],si                ; write return value
        mov     [bp+STACK_USER_ES],es
        mov     es:[si].TPS_QUEUE,0ffffh        ; mark end of list
        mov     bx,[bx].CBL_LINK                ; get next CBL
        jmp     txc_cbl_loop                    ; loop through them

txc_got_some:
```

```
; Link the TPS attached to the CBL to the end of the chain.

        mov     ax,[bx].CBL_TPS_OFF             ; get offset
        mov     es:[si].TPS_QUEUE,ax            ; link to previous TPS
        mov     ax,[bx].CBL_TPS_SEL             ; get selector
        mov     es:[si].TPS_QUEUE+2,ax

; Load pointer to this TPS.

        les     si,DWORD PTR [bx].CBL_TPS_OFF   ; load it
        mov     es:[si].TPS_QUEUE,0ffffh        ; mark end of list

        mov     bx,[bx].CBL_LINK                ; get next CBL
        jmp     txc_cbl_loop                    ; loop through them

txc_tps_done:

        mov     p_1st_rdy_cbl,OFFSET p_1st_rdy_cbl ; invalidate pointers
        mov     p_last_rdy_cbl,OFFSET p_last_rdy_cbl

        mov     ax,p_1st_free_cbl               ; set 1st 586 cbl pointer
        mov     p_1st_586_cbl,ax

; Set new states.

        mov     qtx_state,OFFSET qtx_idle
        mov     cx_state,OFFSET cx_idle
        mov     pwq_state,OFFSET pwq_idle
        mov     cna_state,OFFSET cna_idle

        popf                                    ; restore interrupt state
        mov     WORD PTR [bp+STACK_AX],0        ; no error
        ret

net586_tx_chain ENDP

;*****************************************************************************
; Enqueue transmit state machine - qtx_????
;
; This state machine is made up of 3 states.
;
;       1. qtx_idle - transmitter is currently idle.
;
;               The request is processed immediately and handed directly
;               to the 82586 for transmission.  This is the initial state
;               of this state machine.  This state is entered from qtx_ring
;               when all pending user transmits requests are completed
;               which is detected in cna_none_queued.
;
;       2. qtx_ring - new transmits are being queued into transmit ring.
;
;               This is the current state when the 82586 is busy transmitting
;               a previous request and there are no requests in the wait
;               queue.  States qtx_idle and pwq_wait are responsible for
;               activating this state.
;
;       3. qtx_wait - new transmits are being queued into the wait queue.
;
;               This is the current state when the transmit ring has
;               insufficient free resources to hold all transmit requests.
;               All requests are placed on the wait queue while in this
;               state.  State qtx_ring is responsible for activating this
;               state.
;
;****************************************************************************

qtx_state_machine       PROC    NEAR

;****************************************************************************
; qtx_idle - transmitter is idle
;****************************************************************************

        PUBLIC  net586_qtx_idle                 ; symbol for publication
net586_qtx_idle:

qtx_idle:

; Get the first free CBL.  The state assures that all CBLs are currently
; free.

        mov     bx,p_1st_free_cbl               ; get the cbl pointer

; Store pointer to user's transmit packet structure TPS in the CBL.

        mov     [bx].CBL_TPS_OFF,si             ; save pointer to TPS
        mov     [bx].CBL_TPS_SEL,es

; Clear the TPS status field, and pick up the link fields.

        mov     es:[si].TPS_STATUS,0            ; clear the status
        mov     ax,es:[si].TPS_LAST_TBS         ; get the last TBS offset
        mov     last_tbs_off,ax                 ; store it
        mov     ax,es:[si].TPS_LAST_TBS+2       ; get the last TBS selector
        mov     last_tbs_sel,ax
        les     si,DWORD PTR es:[si].TPS_1ST_TBS ; get pointer to first TBS

; Get pointer to first TBD.

        mov     di,p_1st_free_tbd               ; grab the first tbd

; Setup the CBL.

        mov     [bx].CBL_STATUS,0               ; clear the status
        mov     [bx].CBL_COMMAND,CBL_CMD_XMIT+CBL_EL_BIT+CBL_I_BIT
        mov     [bx].CBL_TBD_OFFSET,di          ; pointer to first TBD

; Fill in the tbd.

        push    bx                              ; save the cbl pointer

qtxi_build_tbds:                                ; build the tbds

        mov     ax,es:[si].TBS_SIZE             ; read the size from the TBS
        mov     [di].TBD_STATUS,ax              ; store the size in the TBD
        mov     dx,es:[si].TBS_BUFFER_PTR       ; get the buffer offset
        mov     bx,es:[si].TBS_BUFFER_PTR+2     ; get the buffer selector

        call    map_586_state                   ; call the mapper

        mov     [di].TBD_ADDR_LO,dx             ; store the address..
```

The reason to do this new fix was based on the observation that whenever the 586 die, it dies because of an overwrite of unfinished command.

And since this is the ONLY place that has the possibility to overwrite the command word of 586 (it does not check if the command has been cleared). I thought the 'NEW FIX' would work, but....

```
        mov     [di].TBD_ADDR_HI,bx             ; ..in the tbd

        cmp     si,last_tbs_off                 ; check for last tbs?
        jne     qtxi_next_tbs                   ; not last, do another
        mov     ax,es                           ; copy selector to ax
        cmp     ax,last_tbs_sel                 ; check selector too
        jne     qtxi_next_tbs                   ; not last, do another

; This is the last tbs.  Set the end of frame bit in the TBD and continue on.

        or      [di].TBD_STATUS,TBD_EOF_BIT     ; set the end of frame bit
        jmp     qtxi_tbds_done                  ; continue on

qtxi_next_tbs:                                  ; advance to the next tbs
        les     si,DWORD PTR es:[si].TBS_LINK   ; point es:si to new tbs

        cmp     di,p_last_free_tbd              ; is this the last tbd?
        je      qtxi_too_many_frags             ; yes, too many fragments

        mov     di,[di].TBD_LINK                ; follow the link
        jmp     qtxi_build_tbds                 ; keep looping

qtxi_too_many_frags:                            ; cleanup and return error

        pop     bx                              ; restore cbl pointer

        mov     WORD PTR [bp+STACK_AX],NET_FRAG_ERROR ; set the return value
        jmp     net586_qtx_return               ; return

qtxi_tbds_done:

        pop     bx                              ; restore the cbl pointer

;
; NEW FIX --See the back of previous page
;       although this is perceived as fix for the real cause of the problem,
;       and it did elongate the up time (without timeout), it inavitably
;       introduces, or unvails, new and fetal bug(s) which will crash both
;       the transmit and receive in about 20 minutes with 80% net traffic
;
;        push    cx
;public patch
;patch:
;       mov     cx,0ffffh                       ; long timeout
;txs_wait_cmd:
;       cmp     scb.SCB_COMMAND,0               ; command clear?
;       je      txs_cmd_clear                   ; yes
;       loop    txs_wait_cmd
;
;txs_cmd_clear:
;        pop     cx
;
; END NEW FIX ---------------------------------
;

; Setup the SCB and get the 586 started.

        mov     scb.SCB_CBL_OFFSET,bx           ; pass cbl pointer
        mov     scb.SCB_COMMAND,SCB_CUC_START   ; start the command unit
        CHANNEL_ATTENTION                       ; alert the 586

        mov     tx_time_count,0                 ; clear the timeout counter
        mov     tx_timeout_enable,0ffffh        ; enable the timeout

; Set new states.

        mov     qtx_state,OFFSET qtx_ring       ; in queueing to ring state
        mov     cx_state,OFFSET cx_transmitting ; transmit is pending
        mov     cna_state,OFFSET cna_none_queued ; nothing more queued

        mov     ax,[bx].CBL_LINK                ; advance pointer to next
        mov     p_1st_free_cbl,ax               ; currently free
        mov     p_last_rdy_cbl,OFFSET p_last_rdy_cbl ; mark none ready

; Assuming there was more than one cbl, the last one couldn't have been used.

        mov     p_1st_586_cbl,bx                ; now belongs to the 586

; Adjust tbd pointer.

        mov     ax,[di].TBD_LINK                ; advance the pointer
        mov     p_1st_free_tbd,ax               ; store it

                                                ; di pointing to last used tbd
        cmp     di,p_last_free_tbd              ; was last tbd used?
        jne     qtxi_return                     ; no, return
        mov     p_1st_free_tbd,OFFSET p_1st_free_tbd ; mark point no good

qtxi_return:

        mov     WORD PTR [bp+STACK_AX],0        ; set return status
        jmp     net586_qtx_return

;*****************************************************************************
; qtx_ring - queueing to CBL and TBD rings
;*****************************************************************************

        PUBLIC  net586_qtx_ring                 ; symbol for publication
net586_qtx_ring:

qtx_ring:

; Get the first free CBL if there is a free one.

        mov     bx,p_1st_free_cbl               ; get the cbl pointer
        cmp     bx,OFFSET p_1st_free_cbl        ; points to self if no more
        jne     qtxr_got_cbl                    ; got a cbl, continue
        jmp     qtxr_wait                       ; no cbl, put on wait queue

; Store pointer to user's transmit packet structure TPS in the CBL.

qtxr_got_cbl:

        push    bx                              ; save the cbl pointer

        mov     [bx].CBL_TPS_OFF,si             ; save pointer to TPS
        mov     [bx].CBL_TPS_SEL,es

; Clear the TPS status field, and pick up the link fields.
```

```
        mov     es:[si].TPS_STATUS,0            ; clear the status
        mov     ax,es:[si].TPS_LAST_TBS         ; get the last TBS offset
        mov     last_tbs_off,ax                 ; store it
        mov     ax,es:[si].TPS_LAST_TBS+2       ; get the last TBS selector
        mov     last_tbs_sel,ax
        les     si,DWORD PTR es:[si].TPS_1ST_TBS ; get pointer to first TBS

; Get pointer to first TBD.

        mov     di,p_1st_free_tbd               ; grab the first tbd
        cmp     di,OFFSET p_1st_free_tbd        ; is there one?
        je      qtxr_tbd_shortage               ; nope

; Setup the CBL.

        mov     [bx].CBL_STATUS,0               ; clear the status
        mov     [bx].CBL_COMMAND,CBL_CMD_XMIT+CBL_I_BIT ; may not be last cbl
        mov     [bx].CBL_TBD_OFFSET,di          ; pointer to first TBD

qtxr_build_tbds:                                ; build the tbds

        mov     ax,es:[si].TBS_SIZE             ; read the size from the TBS
        mov     [di].TBD_STATUS,ax              ; store the size in the TBD
        mov     dx,es:[si].TBS_BUFFER_PTR       ; get the buffer offset
        mov     bx,es:[si].TBS_BUFFER_PTR+2     ; get the buffer selector

        call    map_586_state                   ; call the mapper

        mov     [di].TBD_ADDR_LO,dx             ; store the address..
        mov     [di].TBD_ADDR_HI,bx             ; ..in the tbd

        cmp     si,last_tbs_off                 ; check for last tbs?
        jne     qtxr_next_tbs                   ; not last, do another
        mov     ax,es                           ; copy selector to ax
        cmp     ax,last_tbs_sel                 ; check selector too
        jne     qtxr_next_tbs                   ; not last, do another

; This is the last tbs.  Set the end of frame bit in the TBD and continue on.

        or      [di].TBD_STATUS,TBD_EOF_BIT     ; set the end of frame bit
        jmp     qtxr_tbds_done                  ; continue on

qtxr_next_tbs:                                  ; advance to the next tbs
        les     si,DWORD PTR es:[si].TBS_LINK   ; point es:si to new tbs

        cmp     di,p_last_free_tbd              ; is this the last tbd?
        je      qtxr_tbd_shortage               ; yes, tbd shortage

        mov     di,[di].TBD_LINK                ; follow the link
        jmp     qtxr_build_tbds                 ; keep looping

qtxr_tbd_shortage:                              ; out of tbds, cleanup

        pop     bx                              ; restore cbl pointer

        les     si,DWORD PTR [bx].CBL_TPS_OFF   ; get pointer back to tps

qtxr_wait:                                      ; put request on wait queue
```

```
        mov     p_1st_tx_wait,si                ; put request on wait queue
        mov     p_1st_tx_wait+2,es

        mov     p_last_tx_wait,si               ; it is the only one
        mov     p_last_tx_wait+2,es

; Set new states.

        mov     es:[si].TPS_STATUS,0            ; clear the tps status

        mov     qtx_state,OFFSET qtx_wait       ; put queuer into wait mode
        mov     pwq_state,OFFSET pwq_wait       ; transmits in wait queue

        mov     WORD PTR [bp+STACK_AX],0        ; set the return value
        jmp     net586_qtx_return               ; return

qtxr_tbds_done:
        pop     bx                              ; restore the cbl pointer

; Set new states.

        mov     cna_state,OFFSET cna_ring       ; transmits waiting in ring

; Adjust pointers.

        mov     ax,[bx].CBL_LINK                ; advance pointer to next
        mov     p_1st_free_cbl,ax
        cmp     p_last_rdy_cbl,OFFSET p_last_rdy_cbl ; any ready now?
        jne     qtxr_cbls_rdy                   ; yes, some are ready
        mov     p_1st_rdy_cbl,bx                ; this is first one ready

qtxr_cbls_rdy:
        mov     p_last_rdy_cbl,bx               ; one just done is now ready

; Check to see if last cbl was used.

        cmp     ax,p_1st_586_cbl                ; last cbl used?
        jne     qtxr_chck_tbd                   ; no, check tbds
        mov     p_1st_free_cbl,OFFSET p_1st_free_cbl ; all used, invalidate pointer

qtxr_chck_tbd:

; Adjust tbd pointer.

        mov     ax,[di].TBD_LINK                ; advance the pointer
        mov     p_1st_free_tbd,ax               ; store it

        cmp     di,p_last_free_tbd              ; was last tbd used?
        jne     qtxr_return                     ; no, return
        mov     p_1st_free_tbd,OFFSET p_1st_free_tbd ; mark point no good

qtxr_return:

        mov     WORD PTR [bp+STACK_AX],0        ; set return status
        jmp     net586_qtx_return

;**************************************************************************
; qtx_wait - queueing to transmit resource wait queue
;**************************************************************************
```

```
        PUBLIC  net586_qtx_wait                  ; symbol for publication
net586_qtx_wait:

qtx_wait:

; Put the request at the end of the wait queue.

        mov     bx,p_last_tx_wait                ; pick up last pointer
        mov     ax,p_last_tx_wait+2

        mov     p_last_tx_wait,si                ; update pointer
        mov     p_last_tx_wait+2,es

; Make the last one on the queue point to the new one.

        mov     cx,es                            ; copy new selector
        mov     es,ax                            ; load selector last on queue
        mov     es:[bx].TPS_QUEUE,si             ; set offset
        mov     es:[bx].TPS_QUEUE+2,cx           ; set selector

        mov     WORD PTR [bp+STACK_AX],0         ; set return status
        jmp     net586_qtx_return

qtx_state_machine       ENDP

;****************************************************************************
; Command (Transmit) Not Active state machine - cna_????
;
; This state machine is made up of 3 states.
;
;       1. cna_idle - transmitter is currently idle.
;
;               Does nothing.  This is the initial state.  This state is
;               activated by cna_none_queued.
;
;       2. cna_none_queued - transmitter is active, nothing waiting in ring
;
;               Switches states to cna_idle and qtx_idle when the 82586
;               finishes the last transmit requeust.  This state is activated
;               by qtx_idle and cna_ring.
;
;       3. cna_ring - transmitter active, transmits waiting in ring
;
;               Starts 82586 transmitting next set of transmits when previous
;               set has finished.  This state is activated by qtx_ring and
;               pwq_wait.
;
;****************************************************************************

cna_state_machine       PROC    NEAR

;****************************************************************************
; cna_idle - transmitter is idle
;****************************************************************************

cna_idle        EQU     net586_cna_return        ; idle is a do nothing state

;****************************************************************************
; cna_none_queued - transmit pending but nothing new queued
;****************************************************************************
```

```
        PUBLIC  net586_cna_none_queued          ; symbol for publication
net586_cna_none_queued:

cna_none_queued:

; Set new states and return.

        mov     cna_state,OFFSET cna_idle       ; nothing pending
        mov     qtx_state,OFFSET qtx_idle       ; queuer to idle state
        jmp     net586_cna_return               ; return

;*****************************************************************************
; cna_ring - transmit pending, more queued to transmit ring, none waiting
;*****************************************************************************

        PUBLIC  net586_cna_ring                 ; symbol for publication
net586_cna_ring:

cna_ring:

        mov     bx,p_last_rdy_cbl               ; get pointer to last ready

; Set EL bit in last CBL.

        or      [bx].CBL_COMMAND,CBL_EL_BIT     ; mark the end

; Setup the cbl link in the SCB and add start to the new SCB command.

        mov     ax,p_1st_rdy_cbl                ; get first ready cbl
        mov     scb.SCB_CBL_OFFSET,ax           ; pass cbl pointer
        or      new_scb_command,SCB_CUC_START   ; start command

        mov     tx_time_count,0                 ; clear timeout counter
        mov     tx_timeout_enable,0ffffh        ; enable transmit timeout

; Set cx state to transmitting.

        mov     cx_state,OFFSET cx_transmitting ; transmitter is running

        mov     ax,[bx].CBL_LINK                ; next cbl
        cmp     ax,p_1st_rdy_cbl                ; full circle?
        je      cnar_none_free                  ; no free buffers
        mov     p_1st_free_cbl,ax               ; assume it is free
        cmp     ax,p_1st_586_cbl                ; does 586 own it?
        jne     cnar_free_set                   ; no, continue

cnar_none_free:
        mov     p_1st_free_cbl,OFFSET p_1st_free_cbl ; none free

cnar_free_set:

; Update list pointers.

        mov     p_1st_rdy_cbl,OFFSET p_1st_rdy_cbl ; invalidate ready pointers
        mov     p_last_rdy_cbl,OFFSET p_last_rdy_cbl

; Check to see if state has anything is waiting.

        cmp     pwq_state,OFFSET pwq_wait       ; anything waiting?
        je      cnar_state_set                  ; yes, don't change state
        mov     cna_state,OFFSET cna_none_queued ; no more queued

cnar_state_set:
        jmp     net586_cna_return

cna_state_machine       ENDP

;****************************************************************************
; Process Wait Queue state machine - pwq_????
;
; This state machine is run on the transmit complete interrupt after the
; cx (transmit complete) state machine has been run.  It is designed to pull
; waiting transmits in off of the wait queue and get them into the transmit
; ring as resources become available (transmits complete).
;
;       1. pwq_idle - no transmit requests are on the wait queue.
;
;               Nothing is done.  This is the initial state.  This state
;               is activated by state pwq_wait.
;
;       2. pwq_wait - transmits are waiting in the wait queue.
;
;               Transmit requests are transferred from the wait queue to the
;               transmit ring as ring resources become available.  This
;               state activated by qtx_ring.
;
;****************************************************************************

pwq_state_machine       PROC    NEAR

;****************************************************************************
; pwq_idle - nothing is in the wait queue
;****************************************************************************

pwq_idle        EQU     net586_pwq_return       ; do nothing state

;****************************************************************************
; pwq_wait - transmits waiting on wait queue
;****************************************************************************

        PUBLIC  net586_pwq_wait                 ; symbol for publication
net586_pwq_wait:

pwq_wait:

; Get a CBL.

        mov     bx,p_1st_free_cbl               ; get first free CBL
        cmp     bx,OFFSET p_1st_free_cbl        ; are any free?
        jne     pwqw_get_waiter                 ; yes, got one
        jmp     pwqw_wait1                      ; no, still resource waiting

pwqw_get_waiter:                                ; get first waiting packet

        les     si,DWORD PTR p_1st_tx_wait      ; load pointer

; If no CBL's are currently ready, set the pointer to the 1st ready to
; this one.  Use the pointer to the last ready cbl to indicate if this
; has been done.  If it doesn't change when we get through processing
; then there are no ready cbls.

        cmp     p_1st_rdy_cbl,OFFSET p_1st_rdy_cbl ; any ready?
        jne     pwqw_queue_ring                    ; yes, don't change pointer
        mov     p_1st_rdy_cbl,bx                   ; set the pointer

; Attempt to queue the packet to the transmit ring.

pwqw_queue_ring:

        push    bx                                 ; save the cbl pointer

; Store pointer to user's transmit packet structure TPS in the CBL.

        mov     [bx].CBL_TPS_OFF,si                ; save pointer to TPS
        mov     [bx].CBL_TPS_SEL,es

; Pick up the TPS link fields.

        mov     ax,es:[si].TPS_LAST_TBS            ; get the last TBS offset
        mov     last_tbs_off,ax                    ; store it
        mov     ax,es:[si].TPS_LAST_TBS+2          ; get the last TBS selector
        mov     last_tbs_sel,ax
        les     si,DWORD PTR es:[si].TPS_1ST_TBS ; get pointer to first TBS

        mov     fragment_count,1                   ; keep count of fragments

; Get pointer to first TBD.

        mov     di,p_1st_free_tbd                  ; grab the first tbd
        cmp     di,OFFSET p_1st_free_tbd           ; is there one?
        je      pwqw_tbd_shortage                  ; nope

; Setup the CBL.

        mov     [bx].CBL_STATUS,0                  ; clear the status
        mov     [bx].CBL_COMMAND,CBL_CMD_XMIT+CBL_I_BIT ; may not be last cbl
        mov     [bx].CBL_TBD_OFFSET,di             ; pointer to first TBD

pwqw_build_tbds:                                   ; build the tbds

        mov     ax,es:[si].TBS_SIZE                ; read the size from the TBS
        mov     [di].TBD_STATUS,ax                 ; store the size in the TBD
        mov     dx,es:[si].TBS_BUFFER_PTR          ; get the buffer offset
        mov     bx,es:[si].TBS_BUFFER_PTR+2        ; get the buffer selector

        call    map_586_state                      ; call the mapper

        mov     [di].TBD_ADDR_LO,dx                ; store the address..
        mov     [di].TBD_ADDR_HI,bx                ; ..in the tbd

        cmp     si,last_tbs_off                    ; check for last tbs?
        jne     pwqw_next_tbs                      ; not last, do another
        mov     ax,es                              ; copy selector to ax
        cmp     ax,last_tbs_sel                    ; check selector too
        jne     pwqw_next_tbs                      ; not last, do another

; This is the last tbs.  Set the end of frame bit in the TBD and continue on.

        or      [di].TBD_STATUS,TBD_EOF_BIT        ; set the end of frame bit
        jmp     pwqw_tbds_done                  ; continue on

pwqw_next_tbs:                                  ; advance to the next tbs
        les     si,DWORD PTR es:[si].TBS_LINK   ; es:si now points to new tbs

        inc     fragment_count                  ; got another fragment

        cmp     di,p_last_free_tbd              ; is this the last tbd?
        je      pwqw_tbd_shortage               ; yes, tbd shortage

        mov     di,[di].TBD_LINK                ; follow the link
        jmp     pwqw_build_tbds                 ; keep looping

pwqw_tbd_shortage:                              ; out of tbds, cleanup

        pop     bx                              ; restore cbl pointer

        les     si,DWORD PTR [bx].CBL_TPS_OFF   ; get pointer back to tps

        cmp     fragment_count,NET586_N_TBD     ; more fragments than TBDs?
        jbe     pwqw_wait                       ; no
        jmp     pwqw_fragment_error             ; some still waiting

pwqw_tbds_done:
        pop     bx                              ; restore the cbl pointer
        mov     p_last_rdy_cbl,bx               ; CBL just done is now ready

; Adjust 1st tbd pointer.

        mov     ax,[di].TBD_LINK                ; advance the pointer
        mov     p_1st_free_tbd,ax               ; store it

        cmp     di,p_last_free_tbd              ; was last tbd used?
        jne     pwqw_chck_next                  ; no
        mov     p_1st_free_tbd,OFFSET p_1st_free_tbd ; mark point no good

pwqw_chck_next:

; Pickup the pointer to the TPS out of the CBL.

        mov     ax,[bx].CBL_TPS_SEL             ; selector
        mov     si,[bx].CBL_TPS_OFF             ; offset

; See if another packet is on the wait queue.  Must compare both selector and
; offset with end of queue marker.

        cmp     si,p_last_tx_wait               ; was this last?
        je      pwqw_chk_selector               ; maybe, keep checking
        jmp     pwqw_do_another                 ; no, do another

pwqw_chk_selector:
        cmp     ax,p_last_tx_wait+2             ; was this last?
        je      pwqw_queue_empty                ; yes

pwqw_do_another:

; Pick up the next queued item.

        mov     es,ax                           ; load the selector
        les     si,DWORD PTR es:[si].TPS_QUEUE  ; get new pointer

; Get another CBL.

        mov     bx,[bx].CBL_LINK                ; advance pointer
        mov     p_1st_free_cbl,bx               ; assume it is free
        cmp     bx,p_1st_586_cbl                ; 586 own this one?
        je      pwqw_cbls_out                   ; yes, out of cbls
        jmp     pwqw_queue_ring                 ; no, queue to ring

pwqw_cbls_out:                                  ; ran out of cbls
        mov     p_1st_free_cbl,OFFSET p_1st_free_cbl ; invalidate pointer

pwqw_wait:                                      ; leave request on wait queue

        mov     p_1st_tx_wait,si                ; leave request on wait queue
        mov     p_1st_tx_wait+2,es

pwqw_wait1:

; States don't change.  Check to see if any CBLs are ready.  No CBLs are
; ready if the last ready pointer points to itself.

        cmp     p_last_rdy_cbl,OFFSET p_last_rdy_cbl ; any ready?
        jne     pwqw_some_rdy                   ; yes

        mov     p_1st_rdy_cbl,OFFSET p_1st_rdy_cbl ; no, invalidate pointer
        jmp     net586_pwq_return

pwqw_some_rdy:
        mov     cna_state,OFFSET cna_ring       ; transmits waiting in ring
        jmp     net586_pwq_return               ; all done, return

pwqw_queue_empty:                               ; wait queue is now empty

; No more are waiting.  Set new states.

        mov     pwq_state,OFFSET pwq_idle       ; no more queued
        mov     qtx_state,OFFSET qtx_ring       ; queue new stuff to the ring
        mov     cna_state,OFFSET cna_ring       ; request waiting in ring

        mov     ax,[bx].CBL_LINK                ; next cbl
        mov     p_1st_free_cbl,ax               ; assume it is free
        cmp     ax,p_1st_586_cbl                ; does 586 own it?
        jne     pwqw_ptrs_set                   ; no, continue

        mov     p_1st_free_cbl,OFFSET p_1st_free_cbl ; none free

pwqw_ptrs_set:
        jmp     net586_pwq_return               ; all done, return

pwqw_fragment_error:

; A transmit request has been found that has more fragments than there are
; tbds.  This should never happend since the driver should be configured
; with lots of tbds.  But, if it did happen it would lock up the transmitter.
; Any packets that come through here are pushed back up to the user with an
; error so that the transmitter can continue on.  This code can't be entered
; before all preceeding transmits have completed since the fragment counter
; can never exceed the count of tbd's unless all tbds are free.  This will
; insure that the user gets the packet back in sequence with other transmit
; requests.

        mov     es:[si].TPS_STATUS,TPS_COMP_BIT+TPS_UNSPEC_ERR ; complete, error

        push    si
        push    es
        EXECUTE_1_APNDG ll_appendage            ; notify link level
        pop     es
        pop     si

        cmp     ax,0                            ; was packet link levels?
        je      pwqw_skip_app                   ; skip user appendage if so

        push    si
        push    es
        EXECUTE_1_APNDG tx_appendage            ; notify user
        pop     es
        pop     si

pwqw_skip_app:
        mov     ax,es:[si].TPS_QUEUE            ; update pointers...
        mov     p_1st_tx_wait,ax                ; ...incase we start over
        mov     ax,es:[si].TPS_QUEUE+2
        mov     p_1st_tx_wait+2,ax

        cmp     p_last_rdy_cbl,OFFSET p_last_rdy_cbl ; any ready?
        jne     pwqw_frag_rdy                   ; yes

        mov     p_1st_rdy_cbl,OFFSET p_1st_rdy_cbl ; no, invalidate pointer

pwqw_frag_rdy:
        mov     ax,es                           ; copy selector to ax

; See if another packet is on the wait queue.  Must compare both selector and
; offset with end of queue marker.  If nothing is waiting, must set new states
; and return.

        cmp     si,p_last_tx_wait               ; was this last?
        je      pwqw_frag_selector              ; maybe, keep checking
        jmp     pwq_wait                        ; no, start over

pwqw_frag_selector:
        cmp     ax,p_last_tx_wait+2             ; was this last?
        je      pwqw_frag_exit                  ; yes
        jmp     pwq_wait                        ; no, start over

pwqw_frag_exit:
        mov     pwq_state,OFFSET pwq_idle       ; nothing in queue
        jmp     net586_pwq_return

pwq_state_machine       ENDP

;******************************************************************************
; Command (Transmit) Complete state machine - cx_????
;
; This state machine is made up of 2 states.
;
;       1. cx_idle - transmitter is currently idle.
;
```

```
;               Nothing is done in this state.  This is the initial state.
;               States cx_transmitting and the transmit timeout routine
;               activate this state.
;
;       2. cx_transmitting - transmitter is currently transmitting.
;
;               Transmits are passed back to the user as they complete.
;               The transmit status is set appropriately.  States qtx_idle
;               and cna_ring are responsible for activating this state.
;
;*****************************************************************************

cx_state_machine        PROC    NEAR

;*****************************************************************************
; cx_idle - transmitter currently idle
;*****************************************************************************

cx_idle         EQU     net586_cx_return        ; do nothing state

;*****************************************************************************
; cx_transmitting - transmitter currently active state
;*****************************************************************************

        PUBLIC  net586_cx_transmitting          ; symbol for publication
net586_cx_transmitting:

cx_transmitting:                                ; transmitter is transmitting

        mov     bx,p_1st_586_cbl                ; get pointer to cbl

cxt_check:                                      ; check the cbl status

        mov     cx,[bx].CBL_STATUS              ; get the status

        test    cx,CBL_C_BIT                    ; check completion bit
        jnz     cxt_return_tbds                 ; complete, handle it
        jmp     cxt_return                      ; not complete, exit

; Free up the tbds.

cxt_return_tbds:

        mov     di,[bx].CBL_TBD_OFFSET          ; get pointer to tbd

; If there were no free tbds then make this into a free tbd.

        cmp     p_1st_free_tbd,OFFSET p_1st_free_tbd ; any free?
        jne     cxt_tbd_loop                    ; yes
        mov     p_1st_free_tbd,di               ; reset the pointer

cxt_tbd_loop:

        mov     p_last_free_tbd,di              ; free this one up
        test    [di].TBD_STATUS,TBD_EOF_BIT     ; last one in the chain?
        jnz     cxt_tbds_returned               ; all tbds have been returned
        mov     di,[di].TBD_LINK                ; link to next one
        jmp     cxt_tbd_loop                    ; loop until got 'em all

cxt_tbds_returned:
```

```
; Determine completion status.

        mov     ax,TPS_COMP_BIT                 ; build status in ax
        mov     dx,TPS_OK_BIT                   ; dx cleared if error found

        cmp     crs_check_flag,0                ; CRS checking?
        je      cxt_c2                          ; nope

        test    cx,CBL_S10_BIT                  ; check carrier sense
        jz      cxt_c2
        or      ax,TPS_NO_CRS_BIT               ; set no carrier bit
        xor     dx,dx                           ; found an error

cxt_c2:
        cmp     sqe_check_flag,0                ; SQE checking?
        je      cxt_c3                          ; no SQE checking

        test    cx,CBL_S6_BIT                   ; check SQE bit, should be set
        jnz     cxt_c3                          ; is set, no error

        test    cx,0fh                          ; any collisions?
        jnz     cxt_c3                          ; yes, S6 bit meaningless

        or      ax,TPS_SQE_BIT                  ; signal quality error
        xor     dx,dx                           ; found an error

cxt_c3:
        test    cx,CBL_OK_BIT                   ; error free completion?
        jnz     cxt_c4                          ; yes

        mov     dx,TPS_UNSPEC_ERR               ; assume unspecified error

        test    cx,CBL_S5_BIT                   ; excessive collisions bit
        jz      cxt_c4
        or      ax,TPS_CLSN_BIT                 ; set the bit
        xor     dx,dx                           ; found an error

cxt_c4:

        test    cx,CBL_S8_BIT                   ; DMA underrun?
        jz      cxt_c5
        or      ax,TPS_UNDER_BIT                ; underrun bit
        xor     dx,dx                           ; found an error

cxt_c5:
        add     generic_stats.STS_TX_COUNT,1    ; increment transmit count
        adc     generic_stats.STS_TX_COUNT+2,0

        or      ax,dx                           ; or in other bits
        les     si,DWORD PTR [bx].CBL_TPS_OFF   ; get new pointer
        mov     es:[si].TPS_STATUS,ax           ; write the status

; Execute the transmit complete appendage, if one is setup. First, give
; link level a chance at the packet. If the packet was sent by link level,
; it will return 0 in AX. If this is so, the user appendage should not be
; executed.

        push    bx                              ; save bx
        EXECUTE_1_APNDG ll_appendage            ; execute the appendage
```

The fix here is in effect of NX6.1x .
It is virtually a kick on 586 (an interrupt)
since we (80286) issued command(s) but
no completion interrupt coming back from
586 .

The assumption was 586 didn't see the
command(s) after the 1st one . and we
simply tell 586 again .

The clean up logic is ignored completely.
Since it really does work .

```
        pop     bx                              ; restore bx

        cmp     ax,0                            ; was this link level's packet
        je      cxt_skip_app                    ; skip user appendage if so

        push    bx                              ; save bx
        EXECUTE_1_APNDG tx_appendage            ; execute the appendage
        pop     bx                              ; restore bx

cxt_skip_app:

; If no cbl's are free, this one becomes the 1st free one.

        cmp     p_1st_free_cbl,OFFSET p_1st_free_cbl ; any free?
        jne     cxt_advance_ptr                 ; yes
        mov     p_1st_free_cbl,bx               ; set new free pointer

cxt_advance_ptr:
        mov     bx,[bx].CBL_LINK                ; advance to next cbl

        cmp     bx,p_1st_free_cbl               ; was this the last transmit?
        je      cxt_enter_idle                  ; yes, no more are queued

        cmp     bx,p_1st_rdy_cbl                ; hit the ready list
        je      cxt_enter_idle

        cmp     bx,p_1st_586_cbl                ; gone full circle?
        je      cxt_enter_idle

        jmp     cxt_check                       ; check if new one is complete

; Transmitter empty, update state machines.

cxt_enter_idle:
        mov     cx_state,OFFSET cx_idle         ; enter idle state

        mov     tx_timeout_enable,0             ; disable the timeout

cxt_return:
        mov     tx_time_count,0                 ; clear the timeout counter

        mov     p_1st_586_cbl,bx                ; will be first one next time
        jmp     net586_cx_return                ; jump to return location

cx_state_machine        ENDP

;******************************************************************************
; net586_xmit_timeout - transmitter timeout handler
;
; This module just removes a timed-out transmit from the 82586 transmit ring
; and returns it to the user.  No attempt is made to revive the 586 if it
; has died.
;******************************************************************************

        PUBLIC  net586_xmit_timeout
net586_xmit_timeout     PROC    NEAR

        mov     cx,0ffffh                       ; long timeout
tmout_wait_cmd:
        cmp     scb.SCB_COMMAND,0               ; command clear?
```

see the back of previous page

```
        je      tmout_cmd_clear                 ; yes
        loop    tmout_wait_cmd

tmout_cmd_clear:
        mov     scb.SCB_COMMAND,SCB_CUC_START   ; restart the transmitter
        CHANNEL_ATTENTION                       ; bang on the 586

        mov     tx_time_count, 0
        mov     tx_timeout_enable, 0ffffh
        ret

        mov     bx,p_1st_586_cbl                ; get pointer to cbl

; Free up the tbds.

xt_return_tbds:

        mov     di,[bx].CBL_TBD_OFFSET          ; get pointer to tbd

; If there were no free tbds then make this first free tbd.

        cmp     p_1st_free_tbd,OFFSET p_1st_free_tbd ; any free?
        jne     xt_tbd_loop                     ; yes
        mov     p_1st_free_tbd,di               ; reset the pointer

xt_tbd_loop:

        mov     p_last_free_tbd,di              ; free this one up
        test    [di].TBD_STATUS,TBD_EOF_BIT     ; last one in the chain?
        jnz     xt_tbds_returned                ; all tbds have been returned
        mov     di,[di].TBD_LINK                ; link to next one
        jmp     xt_tbd_loop                     ; loop until got 'em all

xt_tbds_returned:

        les     si,DWORD PTR [bx].CBL_TPS_OFF   ; get tps pointer
        mov     es:[si].TPS_STATUS,TPS_COMP_BIT+TPS_TIMEOUT_BIT ; write status

; Execute the transmit complete appendage, if one is setup. First, give
; link level a chance at the packet. If the packet was sent by link level,
; it will return 0 in AX. If this is so, the user appendage should not be
; executed.

        push    bx                              ; save bx
        EXECUTE_1_APNDG ll_appendage            ; execute the appendage
        pop     bx                              ; restore bx

        cmp     ax,0                            ; was this link level's packet
        je      xt_skip_app                     ; skip user appendage if so

        push    bx                              ; save bx
        EXECUTE_1_APNDG tx_appendage            ; execute the appendage
        pop     bx                              ; restore bx

xt_skip_app:

; If no cbl's are free, this one becomes the 1st free one.

        cmp     p_1st_free_cbl,OFFSET p_1st_free_cbl ; any free?
        jne     xt_advance_ptr                  ; yes
        mov     p_1st_free_cbl,bx               ; set new free pointer

xt_advance_ptr:
        mov     bx,[bx].CBL_LINK                ; advance to next cbl

        cmp     bx,p_1st_free_cbl               ; was this the last transmit?
        je      xt_enter_idle                   ; yes, no more are queued

        cmp     bx,p_1st_rdy_cbl                ; hit the ready list
        je      xt_enter_idle

        cmp     bx,p_1st_586_cbl                ; gone full circle?
        je      xt_enter_idle

        jmp     xt_return                       ; return

xt_enter_idle:

; Transmitter empty, update state machines.
public xt_enter_idle
        mov     cx_state,OFFSET cx_idle         ; enter idle state

        mov     tx_timeout_enable,0             ; disable the timeout

xt_return:
        mov     tx_time_count,0                 ; clear the timeout counter
        mov     p_1st_586_cbl,bx                ; will be first one next time
        ret

net586_xmit_timeout     ENDP

_TEXT           ENDS
                END
```

February 14, 1991

UPDATE NOTE
NX300 Network Executive
Release 6.2

This update note for NX300 Network Executive, Release 6.2 describes the features of Release 6.2 and lists the fixes that have been made since previous releases.

If you should have any questions, comments, or suggestions, please contact:

Customer Service Center
Federal Technology Corporation
207 South Peyton Street
Alexandria, VA 22314
1-800-FON4LAN

FEEDBACK

Your feedback on this product is appreciated. An overall evaluation form, as well as a problem report form, are included at the end of this update note. Please complete the evaluation form and return it to Federal Technology Corporation. Please use the problem report as necessary.

DIFFERENCES BETWEEN NX VERSION 5 AND VERSION 6 IN LINK-LEVEL MODE

The transmit buffering area is now 32 Kbytes. All remaining memory is now used for receive buffering.

NX Version 6 uses approximately 16 Kbytes for internal data area and 32 Kbytes for transmit buffering. All remaining memory space is available for receive buffering. The maximum memory used in any case is limited to 512 Kbytes. Buffers are allocated to hold a maximum size packet. For 1518-byte Ethernet packets the approximate board buffering capability is as follows:

Memory Size	Transmit Buffers	Receive Buffers
128 Kbytes	20	50
256 Kbytes	20	140
512 Kbytes	20	310

* The number of multicast addresses is no longer configurable. Sixty-four multicast slots are always configured.

* I/O-mapped and memory-mapped host interrupts are no longer supported.

* Configuration error codes, fatal error codes, and warning codes have been changed. The error codes for the two versions of the NX firmware are mutually exclusive. Status codes in link-level reply messages remain unchanged.

* The configuration message has been changed. Although the NX Version 6 configuration message is compatible with that of Version 5, several changes have been made. The following fields have become reserved fields in Version 6:

  - Memory Map Size
  - Memory Map
  - NX Movable Block Address
  - Number of Processes
  - Number of Mailboxes
  - Number of Multicast Slots
  - Host-to-EXOS Message Queue Interrupt Value
  - EXOS-to-Host Message Queue Interrupt Value

  Version 6 of the NX firmware ignores these fields. The request and reply values of these fields are undefined.

  The following fields have recommended specified values in the NX Version 6 configuration. Other nonspecified values are supported for compatibility with Version 5 but the use of these values is discouraged. Version 6 provides the best performance if the specified values are used.

  - EXOS Mode = 0
  - Host Data Format Option = 0101H
  - Host Address Mode = 3
  - Number of Hosts = 1

  For the fields Host-to-EXOS Message Queue Interrupt Type and EXOS-to-Host Message Interrupt Type, option 1 (I/O mapped) and option 2 (memory mapped) are no longer valid. Option 3 (level interrupt) and option 4 (bus-vectored interrupt) have been combined into a single option, option 4 (bus hardware interrupt). Option 3 is still supported, but its use is not recommended.

* The NET_TRANSMIT reply no longer returns the slot number the transmit request occurred on. This field in the request/reply message is now reserved and always returns a value of zero.

* The NET_TRANSMIT return code 10H is now a general transmit error that indicates the transmit was unsuccessful.

* A new NET_TRANSMIT request (request code OFH) has been added. This request returns a reply message to the host as soon as the data in the host buffer have been copied to the EXOS board. With this form of message, the transmit status is not returned to the host. Therefore, protocol on the host must detect transmit failures.

* The network statistics counters no longer stick at their maximum value. Instead, they roll over and continue counting.

* The loopback error bit available to the host in the status port is now a generic warning bit. Any warning, including loopback test failure, causes this bit to be set.

* The debug jumper is now the NX mode jumper. When NX PROMs that support both Versions 5 and 6 are used, this jumper selects the mode to run in. Installing the jumper causes Version 6 to run. Removing the jumper causes Version 5 to run.

UPGRADING TO NX 6.2

If you have an EXOS 301, EXOS 302, or EXOS 304 board installed in your system, you must replace both EPROMs on the board before you can use Release 4.0 of the LAN Service network software. The EPROMs you are removing contain Version 5.x of the Network Executive (NX) firmware. The EPROMs you are replacing them with contain Version 6.2 of NX (NX 6.2) as well as the latest NX 5.x firmware for your board (the latter is provided for downward compatibility).

Follow these steps to upgrade to the NX 6.2 EPROM:

1. Power down your system.

2. Remove the EXOS board from your system.

3. Locate the EPROM marked NX300L and make a note of its orientation.

4. Remove the EPROM from the board.

5. Insert the new EPROM (NX300L 6.2) in the board. Make sure it is oriented the same way as the old EPROM.

6. Locate the EPROM marked NX300H and make a note of its orientation.

7. Remove the EPROM from the board.

8. Insert the new EPROM (NX300H 6.2) in the board. Make sure it is oriented the same way as the old EPROM.

9. Jumper the EXOS board for 27256 EPROMs, as follows:

Board	**Jumpers**
EXOS 301	J15 1-2 present
EXOS 302	J15 2-3 present; J16 1-2 present
EXOS 304	J18 1-2 present; J19 1-2 present

10. Insert the NX 6.x select jumper. This jumper selects the operation mode of NX. The EXOS board operates in NX 6.x mode when the jumper is present and in NX 5.x mode when the jumper is absent.

Board	**Jumpers**
EXOS 301	J9 present for NX 6.x mode
EXOS 302	J9 present for NX 6.x mode
EXOS 304	J9 present for NX 6.x mode

11. Re-insert the EXOS board in the system and reconnect the Ethernet cables.

12. Power on the system. Ensure that the status LED on the EXOS board starts flashing at a steady rate approximately 3 seconds after you power on the system. If the status LED remains lit constantly, check that you have replaced the NX EPROMs properly.