Software Driver Programmer's Guide for the DP83932 SONIC™

National Semiconductor
Application Note 746
Wesley Lee
Mike Lui
March 1991

INTRODUCTION

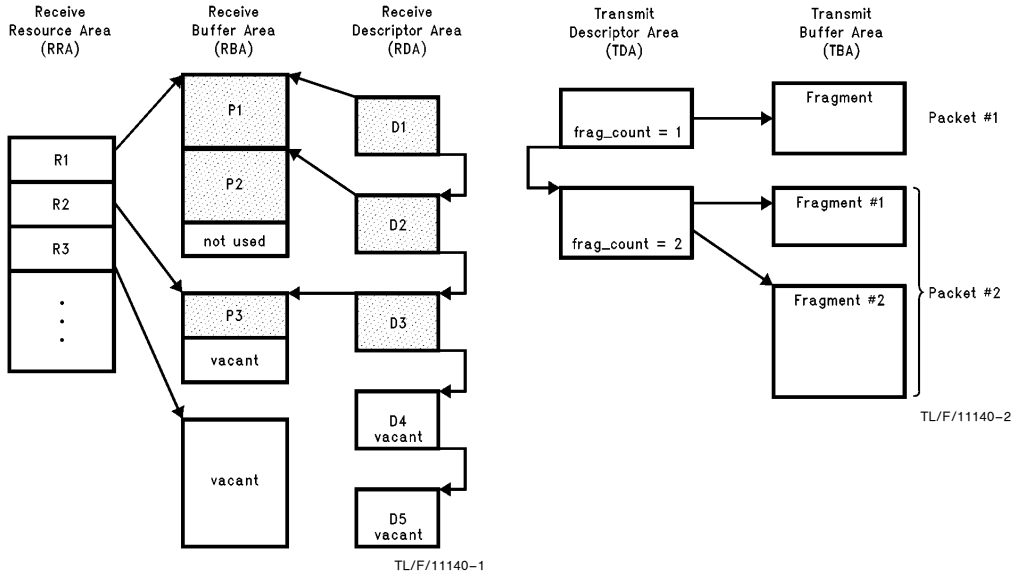
In the past, Ethernet chips have concentrated on interfacing well with the hardware, but have given the software interface only passing notice. While hardware designers may have been satisfied, the software developers were forced to write drivers for unwieldy silicon. Recently, with companies looking for ways to increase performance, they have found that the software interface is crucial and has been one of the bottlenecks in the system. A chip with an over constraining buffer management slows down the system by introducing more levels of indirection (pointers) than are truly needed by the system software. In view of these shortcomings, National surveyed a number of software developers to define a buffer management system which operates efficiently with the driver. Their basic response was *Keep it Simple.* The reasons were twofold. First, a simple software interface engenders a driver which is easy to write and secondly, a simpler, thus shorter, driver leads to a faster driver. The SONIC's buffer management epitomizes this with three salient features. First, only one level of indirection is used to reference data in memory; secondly, link-lists are chosen to endow the software developer with the flexibility to easily manipulate descriptors, and thirdly, a register-based command interface is provided to make commands fast and immediate.

ABOUT THIS GUIDE

This guide will provide you the information needed to write a driver for the DP83932 System-Oriented Network Interface Controller (SONIC). You will first be introduced to basic algorithms using the SONIC's buffer management, then be shown actual implementation examples. It is recommended that you are familiar with the DP83932 SONIC datasheet before reading this document.

1.0 THE DRIVER SOFTWARE—SONIC INTERACTION

The key to making a Driver and all upper levels of the network software efficient, is to ensure that they must be capable of referencing received or transmitted packets via pointers and then conveying these pointers up to the next level of software. By employing pointers in this manner, needless packet copying from one area in memory to another is eliminated. As shown in *Figure 1-1*, the SONIC's descriptor areas, the RDA and TDA reference the received and transmitted packet and the RRA references the buffers for the received packets. The actual received and transmitted packets remain in their original locations in the RBA and TBA and are not copied elsewhere. In this section the basic algorithms are given to illustrate the usage of the RDA, RRA and TDA. Section 4.0 describes the implementation examples.



R = resource descriptor
D = packet descriptor
P = packet

FIGURE 1-1. Overview of the SONIC's Buffer Management

SONIC™ is a trademark of National Semiconductor Corporation.

RRD-B30M75/Printed in U. S. A.

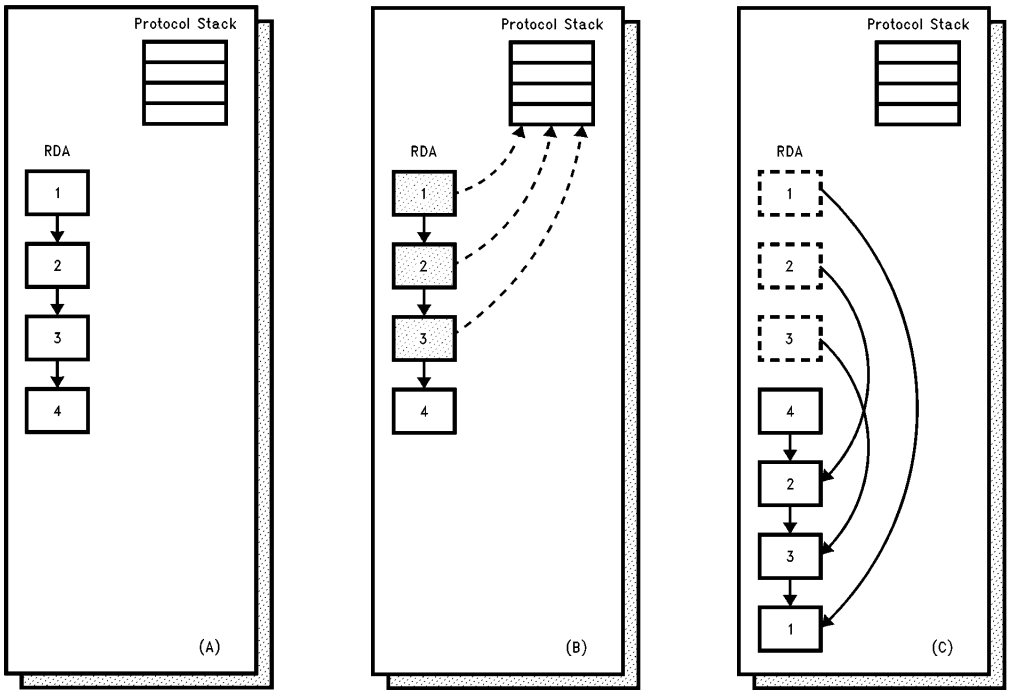
1.1 Processing Packets in the Receive Descriptor Area (RDA)

After the SONIC has received the packet, it places the packet in the RBA and the packet information in the RDA. The Driver, in turn, processes this packet by locating the packet from the packet pointer (RXpkt.ptr0,1) fields in the RDA and then delivering the pointer up to the next level software for further processing. The Driver then returns the descriptor to the front of the list for reuse. This process is illustrated in *Figures 1-2 (a)*, *(b)*, and *(c)*. Note that the link-list allows descriptors to be appended to the front of the list in any order. Note also that no special considerations are required to append receive descriptors. The pseudo code below illustrates the simplicity in appending descriptors.

```
append_descr( )
      new_RXpkt.link = 1;
      old_RXpkt.link = new_RXpkt.status
      /* Old link field points to
      address of new status field */
```

1.2 Recycling Buffers in the Receiver Resource Area (RRA)

Intermixed with processing of packets, the Driver must also replenish the receive buffer pool by adding resources descriptors to the RRA. A suggested method for replenishing receive buffers is given in the following example. (This method assumes that more than one packet is stored in an RBA.)



TL/F/11140–3 TL/F/11140–4 TL/F/11140–5

FIGURE 1-2 a, b, c. Processing Descriptors in the RDA

(a) Initial condition: four descriptors are available for use
(b) Packets received: three packets having been received are then passed up to the upper level software for further processing.
(c) Packets processed: the upper level software having finished processing the packets, the Driver returns the descriptors to the front of the list.

The Driver allocates a fixed number of receive buffers (RBAs), determined at initialization, and recycles them as they are used. When the upper level software receives a packet from the driver (via pointers), it processes the packet at the location received (in the RBA), and when done, notifies the Driver of the freed memory space. The Driver, then, records this event by tallying the packet in a ''scoreboard'' corresponding to the RBA (see *Figure 1-3*). When the number of packets processed equals the total number of packets in an RBA, then RBA is free and may be returned to the RRA ring.

RBA #	Packets Processed	Total Packets in RBA
0	5	5
1	2	4
2	3	6
3	3	Unknown

FIGURE 1-3: RBA Scoreboard Example

RBA#0 is now free since packets processed equals total packets in RBA. For RBA #3, the software does not yet know how many packets reside in an RBA since the SONIC has not finished using this RBA. When the software detects the LPKT bit set, the packet sequence number reveals the total number of packets (see below).

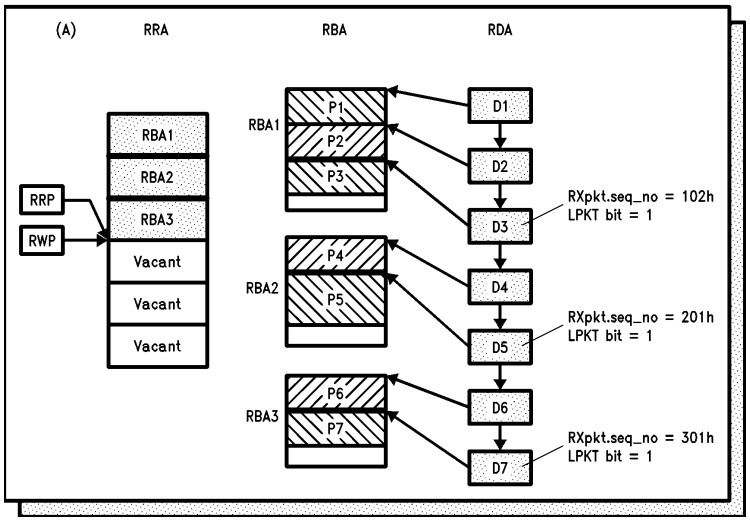
Because packets may be processed in any order (thus, packets may be freed up out of sequence), freeing up an RBA is not a straight forward. However, the SONIC reduces this task to a simple tallying procedure with its Receive Sequence Numbers (RXpkt.seq__no). When the Driver detects the LPKT (last packet) bit set to a 1, the sequence numbers indicate how many packets are in a given RBA. Thus, the Driver simply tallies the number of packets processed for a given RBA and when this is equal to the total number of packets, the RBA is free. The sequence numbers are shown below.

15 — 8	7 — 0
RBA Sequence Number (Modulo 256)	Packet Sequence Number (Modulo 256)

If LPKT = 1

packet sequence number equals total number of packets minus one in the RBA (packet sequence number starts at zero)

The following three figures (*Figures 1-4a*, *1-4b*, and *1-4c*) show a scenario depicting the Driver using 3 RBAs and updating the RBA ''scoreboard''. The flowchart in section 4.2 (*Figure 4-3*) illustrates the recycling of RBAs during receive processing.



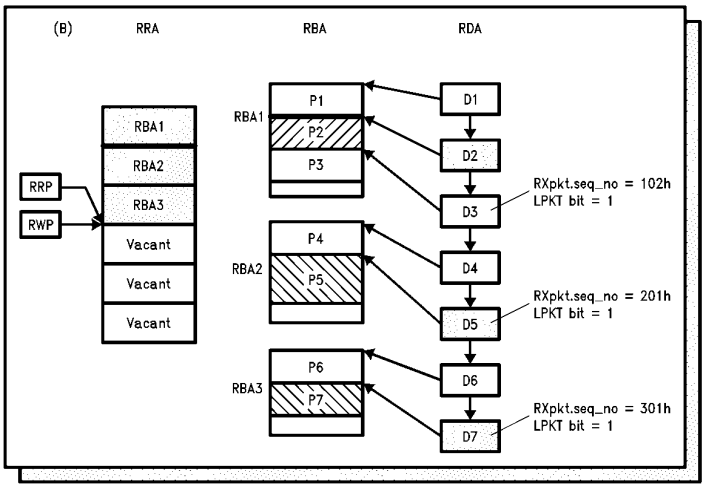
TL/F/11140–6

RBA Scoreboard

RBA #	Processed Packets	Total Packets
1	0	3
2	0	2
3	0	2

FIGURE 1-4 (a). Recycling Buffers in the RRA

(a) This figure shows the SONIC, having stored seven packets (P1–P7) in the RBA, has exhausted all its receive buffers (RRP = RWP). The RBA scoreboard indicates that there are 3 unprocessed packets in RBA #1, 2 in RBA #2 and 2 in RBA #3. These numbers are determined by the RXpkt.seq__no field.



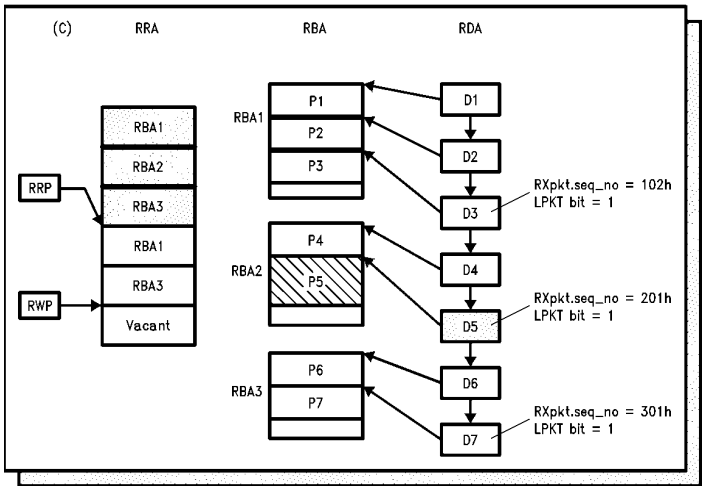
TL/F/11140–7

RBA Scoreboard

RBA #	Processed Packets	Total Packets
1	2	3
2	1	2
3	1	2

FIGURE 1-4(b). Recycling Buffers in the RRA

(b) The upper level software has finished processing four packets (P1, P3, P4, and P6) and has notified the Driver of this action. The RBA scoreboard now indicates that there is 1 unprocessed packet in RBA #1, 1 in RBA #2 and 1 in RBA #3.



TL/F/11140–8

RBA Scoreboard

RBA #	Processed Packets	Total Packets	
1	3	3	→ RBA 1 may be recycled
2	1	2	
3	2	2	→ RBA 3 may be recycled

FIGURE 1-4 (c). Recycling Buffers in the RRA

(c) The upper level has now finished processing 6 packets (P1, P2, P3, P4, P6, and P7). The RBA scoreboard now indicates that RBA #1 and RBA #3 are freed up. The Driver returns these buffers back to the RRA and increments the RWP register accordingly.

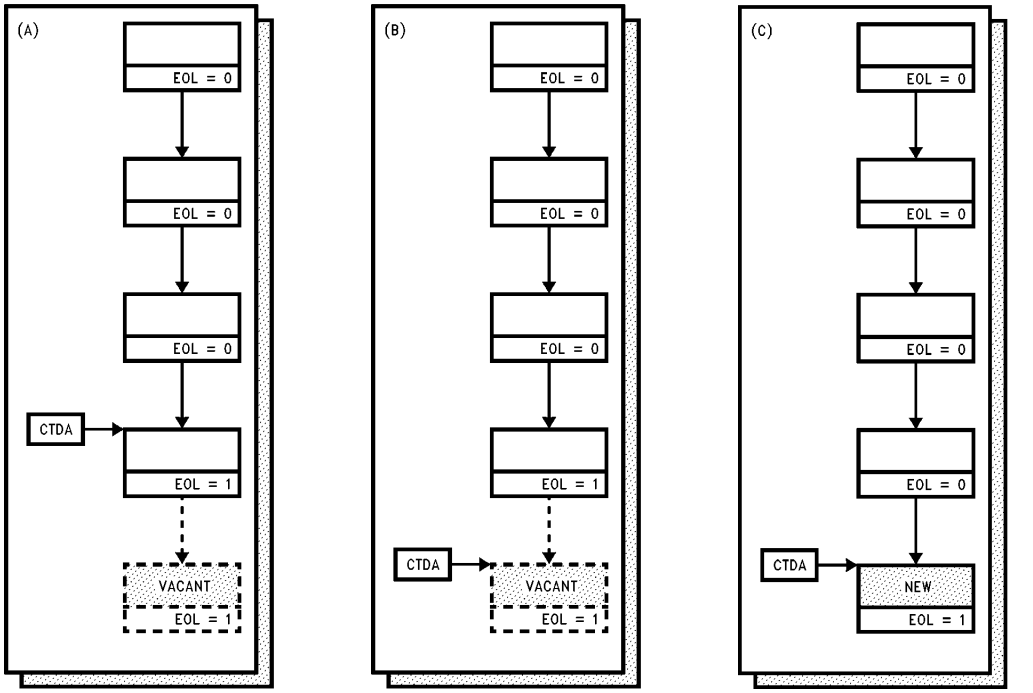
1.3 Transmitting Packets from the Transmit Descriptor Area (TDA)

For transmit operation, the Driver uses the TDA to enqueue packets for transmission. Multiple packets may be sent from a single command with each packet allowed to be fragmented (reside in different areas in memory). The fragments themselves may be as small as 1 byte and begin on any byte boundary. Furthermore, particular attention has been made to allow the Driver to append descriptors ''on the fly''.

To send packets, the driver first creates a list of descriptors in the TDA, then issues the transmit command. The SONIC then reads the TDA and transmits the packets. Once a list is created, the Driver can add to this list ''on the fly'' without the SONIC stopping. The following rule, however, must be followed: *the last TXpkt.link field must point to the next location where a descriptor will be added* as illustrated in *Figure 1-5 (a).* The procedure for appending descriptors is outlined as follows:

1. Create a new descriptor with its TXpkt.link pointing to the next vacant descriptor location and its EOL bit set to a ''1''.
2. Reset the EOL bit to a ''0'' of the previously last descriptor.
3. Re-issue the Transmit command (setting the TXP bit in the Command register).

Re-issuing the Transmit command assures that the SONIC will continue to send all packets in the list. If the SONIC is currently transmitting, the Transmit command has no effect. If the SONIC has stopped transmitting (which occurs if the SONIC has reached the last descriptor before the Driver has had a chance to append to it) it continues transmitting from where it had previously stopped. The rule, as stated above, guarantees that the Current Transmit Descriptor (CTDA) register points to a valid descriptor after the SONIC has stopped transmitting (see *Figures 1-5 (a), (b)* and *(c)*).



TL/F/11140–9 TL/F/11140–10 TL/F/11140–11

FIGURE 1-5 a, b, c. Appending Descriptors ''On the Fly'' in the TDA

These series of figures shows a scenario whereby the SONIC has reached the end of the descriptor list before the Driver has appended a new descriptor.

(a)This figure shows the Driver has created a list of four descriptors with the last descriptor pointing to the next location where a descriptor will be added. The transmit command has subsequently been issued and the SONIC has reached the last descriptor.

(b) The SONIC has finished transmitting the last descriptor. It reads the last link field and updates the CTDA register to point to the vacant descriptor location. Note that the CTDA register is already prepared for the next transmission.

(c) The Driver has appended a descriptor at the vacant location and reissues the transmit command. Note that the CTDA register is pointing to the proper location.

2.0 REGISTER MODEL OF THE SONIC

As a brief review, this section gives a short description of the SONIC User registers. This section is similar to section 4.0 of the SONIC datasheet. It may be skipped without loss of continuity.

2.1 Register Layout

The SONIC contains 64 16-bit registers used for conveying status and control information. Not all registers, however, are needed by the system since some registers are used for internal operations of the SONIC and others used for in-house factory testing. The registers are categorized as follows:

User Registers: The registers are accessed by the user to status, control and monitor SONIC operations. These are the only registers you need to access.

Internal Use Registers: These registers are used by the SONIC during the course of operation and are not intended to be accessed by you.

Factory Test Registers: These registers are used by National Semiconductor for production testing of the SONIC and should not be accessed. Accessing these registers during SONIC operations may cause erratic behavior.

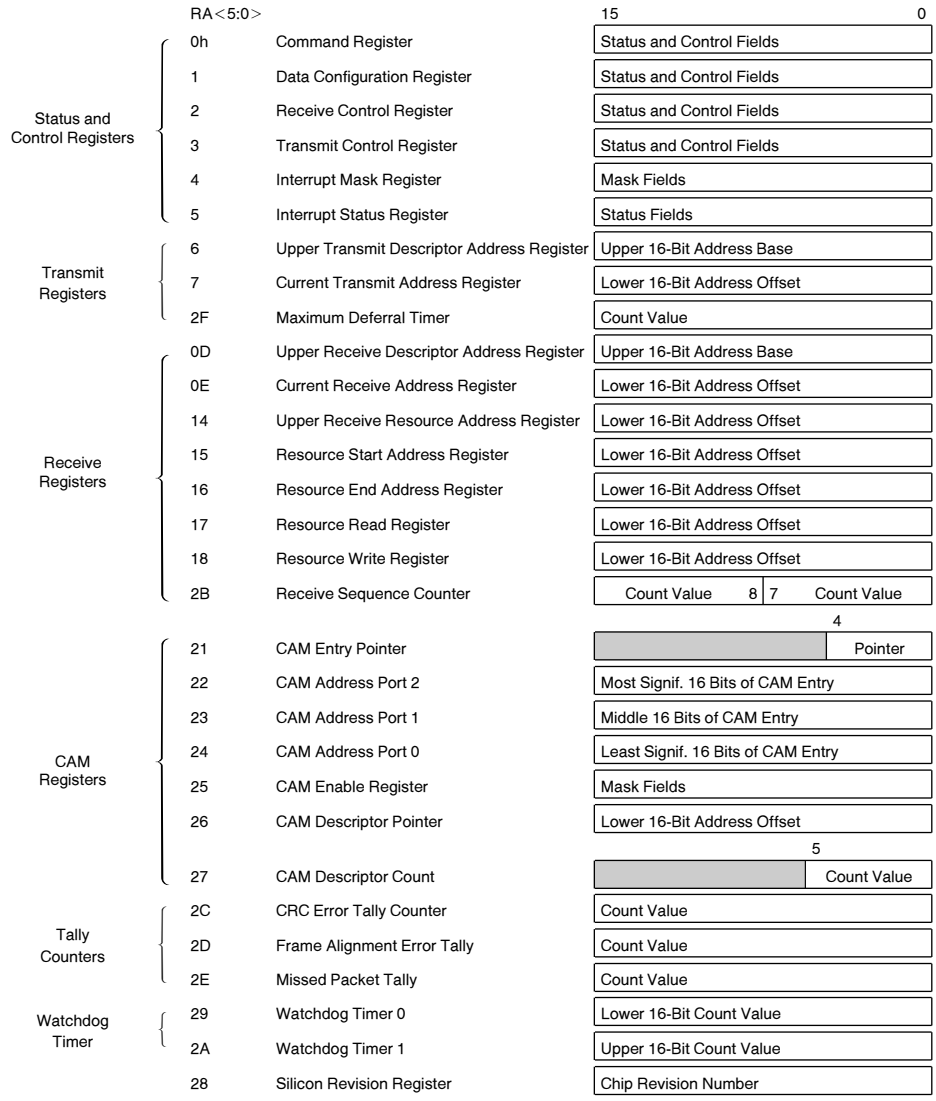


FIGURE 2-1. User Register Grouping

2.2 User Register Grouping

The User register may be further categorized into 6 groups (*Figure 2-1*) based upon their functionality, i.e., Status and Control, Transmit, Receive, Content Addressable Memory (CAM), Tally counters, and General-Purpose timer. These groups are described as follows:

2.2.1 Status and Control Registers

These registers, controlling the transmit, receive, bus, and interrupt operations of the SONIC, consist of the Command, Data Configuration, Receive Control, Transmit Control, Interrupt Mask, and Interrupt Status registers. Of these registers, only the Command and Interrupt Status register are accessed frequently during operation; all others are generally accessed only once during initialization (see section 3.0). These registers are briefly described below.

Command register. This register is used for issuing the commands to the SONIC such as transmitting packets, enabling the receiver, and software reset. Commands may be issued by setting the corresponding bit to a ''1''. During normal operation, the transmit command is the only command that is generally used.

Data Configuration register: This register configures the bus interface circuitry, programming the data width size (16 or 32 bits), wait-state insertion (if any), and FIFO threshold. This register may only be written to while the SONIC is in software reset.

Receive Control register. This register contains two type of bits, configuration and status. The configuration bits program the SONIC to accept the different classes of packets which may be received such as Physical, Multicast, Broadcast packets, and Runt and Errored packets. The SONIC can also accept all packets from the network for network management and Bridge applications. The Receive Control register also reports the status of the received packet. The software should not read this register directly since status is updated from the next incoming packet and the previous status is overwritten. Instead, the software obtains the status in the status field (RXpkt.status) of the Receive Descriptor Area.

Transmit Control register. This register also contains two types of bits, configuration and status. The configuration bits program the various transmit options for (1) generating and interrupts after selected packets have been transmitted, (2) enabling when the ''Out of Window'' collision timer begins (either at the beginning of the packet or at the State of Frame Delimiter), (3) inhibiting the CRC from being appended to the packet, and (4) enabling the excessive deferral timer (3.2 μs). The software should not load this register directly; instead, it writes to the configuration field (TXpkt.config) of the Transmit Descriptor Area (TDA) which the SONIC reads before transmission. The status bits post status of the transmitted packet. Again, this register is not directly read since the SONIC clears the status after it reads the TXpkt.link field. Instead, the software acquires status from the state field (TXpkt.status) in the TDA.

Interrupt Mask register. This register enables the various interrupts that the SONIC may generate. Writing a ''1'' to the bit enables the corresponding interrupt.

Interrupt Status register. This register reports interrupts which the SONIC has generated. Interrupts are indicated by a ''1'' and are cleared when a ''1'' has been written to it. Since writing a ''0'' to any bit has no effect, only the specified bits are cleared during the write operation.

2.2.2 Transmit Register

The Transmit registers, the Upper Transmit Descriptor Address (UTDA) and the Current Transmit Descriptor Address (CTDA) registers, locate the active descriptor in the Transmit Descriptor Area. The UTDA register, containing a fixed upper 16 bits of address, A<31:16> and CTDA register, containing an active lower 15 bits of address, A<15:1> are concatenated together to form a complete 31-bit address. (The SONIC only provides word or double word addressing.) The LSB of the CTDA register is the End of List (EOL) bit and is used by the SONIC to determine the last descriptor in the list.

2.2.3 Receive Registers

The receive registers consist of the Receive Sequence Counter, the End of Buffer Count (EOBC) register, and two groups of registers, the descriptor registers and the resource registers. These registers are briefly described as follows:

The Receive Sequence Counter: This counter indicates the number of packets that reside in a particular Receive Buffer Area (RBA). See section 1.1 for an explanation on how to use this register.

EOBC register. This register defines the lower boundary in the RBA. If after reception, the remaining numbers words in the RBA are equal to or greater than the EOBC register, reception continues within the same RBA; otherwise, the SONIC stores the packet in another RBA.

Descriptor registers: These registers locate the active descriptor in the Receive Descriptor Area (RDA) and are composed of the Upper Receive Descriptor (URDA) and the Current Receive Descriptor (CRDA) registers. These registers are concatenated similarly as the Transmit registers (UTDA and CTDA) above where the URDA contains a fixed upper 16 address bits, A<31:16> and the CRDA contains the lower 15 address bits, A<15:1>. The LSB of the CRDA register is used by the SONIC to determine the last descriptor in the receive list.

Resource registers: These registers, used to define the Receive Resource Area (RRA), composed of the Resource Start Area (RSA), the Resource End Area (REA), Resource Write Pointer (RWP), Resource Read Pointer (RRP) and the Upper Receive Resource Address (URRA) registers. The first two registers are static and define the starting and ending points of the RRA. The second two are active and respectively point to the next location where the software places a new descriptor and where the SONIC reads the next descriptor. The SONIC concatenates the last register, the URRA with the other registers to provide a full 31-bit address. The URRA register contains a fixed upper address, A<31:16> and the other four contain an active lower address, A<15:1>. The LSB of these registers is not used since the SONIC only provide word or double word addressability.

2.2.4 CAM Registers

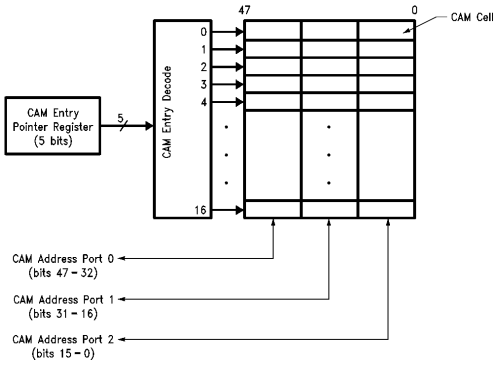
The CAM registers are used to access the 16 48-bit CAM entries. Because random accessibility to all CAM entries would consume too much register space (16 x 3 = 48 locations), the CAM entries are accessed via a 4-bit pointer register (CAM Entry Pointer) and 3 16-bit ports (CAM Access Ports 0 to 2). The CAM Entry Pointer selects 1 of 16 entries and the CAM Access Ports 0 to 2, respectively access the least through the most significant portions of the 48-bit entry *(Figure 2-2)*.

Note: The least significant byte of the address is the *first* byte received/transmitted from the network.

Reading the CAM

The CAM is accessed in the following manner:

1) Place the SONIC in software reset by setting the RST bit in the Command register. This condition must be met before reading the CAM.
2) Select the CAM entry by writing the corresponding value in the CAM Entry Pointer.
3) Read the CAM Address Ports 0 to 2 to obtain the complete 48-bit entry.



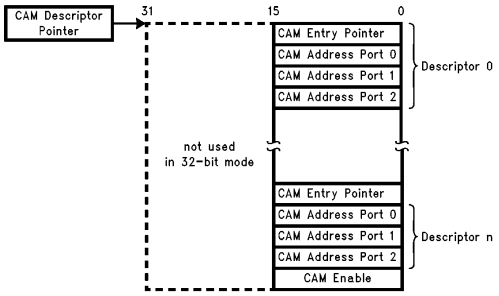
TL/F/11140–12

FIGURE 2-2. CAM Organization

Writing to the CAM

To avoid internal conflicts with the CAM entries when receiving packets, the SONIC does not allow the entries to be written to directly. Instead, the entries are written to indirectly via the CAM Descriptor Area (CDA). This area, maintained in memory, contains the data to be written into the CAM and upon command, the SONIC reads this area and load its CAM. The CDA is composed of *n* number descriptors (*Figure 2-3*) which are used to load the CAM Entry Pointer, the CAM Access Ports, and the CAM Enable register. To program the CAM, you first initialize the CDA, load the CAM Descriptor Count register with the number of descriptors and the CAM Descriptor Pointer register with the starting address of the CDA, then issue the Load CAM command to the SONIC. This operation is summarized below:

1) Load the CDA as specified in *Figure 2-3*.
2) Load the CAM Descriptor Count register with the number of descriptors.
3) Load the CAM Descriptor Pointer register with the starting address of the CDA.
4) Issue the Load CAM command (setting the LCAM bit in the Command register). The SONIC finishes this command when the LCAM bit is reset.



TL/F/11140–13

FIGURE 2-3. CAM Descriptor Area Format

2.2.5 Tally Counters

The Tally counters maintain the network management events which occur too frequently for the software to maintain. These events, CRC errors, frame alignment errors, and missed packets are tallied by the CRC, FAE and Missed Packets Tally counters. these counters are 16-bit counters and can generate an interrupt when a rollover occurs. These registers are generally used in conjunction with software to maintain a 32-bit counter. These counters maintain the time-sensitive lower 16 bits of the count while software maintains the upper 16 bits.

2.2.6 General-Purpose Timer

This 32-bit timer, clocked at one half the 10 MHz transmit clock frequency, is used for timing user definable events. The timer measures events ranging from microseconds up to minutes. The time can be calculated by multiplying the count value by 200 ns (½ the transmit clock period). Table 2-1 gives some example values. To use the timer, you first load the timer with a count value, then start the timer by setting the ST bit in the Command register. The SONIC then begins decrementing the timer. When the rollover is reached (0000 0000h to FFFF FFFFh), the Timer Complete (TC) bit in the Interrupt Status register is set. Note that the timer does not stop when the rollover occurs, but continues to decrement (from FFFF FFFFh). It must be explicitly stopped by setting the STP bit in the Command register.

Table 2-1. Example Timer Values

Timer	WT1	WT0
0.1 sec	7	A120
0.5 sec	26	25A0
1.0 sec	4C	4B4
10 sec	2FA	F080
30 sec	8F0	D180
1 min	11E1	A300
5 min	5968	2F00
10 min	B2D0	2E00

2.2.7 Silicon Revision Register

This register supplies information on the revision stepping of the SONIC. This register begins at zero and counts upward. Contact National Semiconductor for latest information on this register.

3.0 INITIALIZING THE SONIC

Initializing the SONIC is the crucial first step before any SONIC operations can commence. This step involves setting up the SONIC's registers for reception and transmission and initializing the memory structures for the Buffer Management. This section describes the initialization process by introducing what information is needed, then discussing an example initialization routine.

Getting Started

Before initializing the SONIC, a few details regarding the hardware and network operating system must be obtained. By answering the questions below, the required information can be gathered.

1) What is the bus size?

   The SONIC supports bus sizes of 16 or 32 bits.

2) Does the system operate in a synchronous or asynchronous manner?

   This question refers to how the $\overline{\text{RDYi}}$ (or $\overline{\text{DASCK0,1}}$) input is issued to the SONIC. If this line is asserted with guaranteed setup and hold times by the hardware, use synchronous mode; otherwise, use asynchronous mode. Synchronous mode has the advantage of having a minimum memory cycle of 2 bus clocks as opposed to 3 bus clocks for asynchronous mode.

3) What is the maximum bus latency does the SONIC expect?

   The bus latency is the time from when the SONIC requests for the bus (by asserting the HOLD or $\overline{\text{BR}}$ pin) to when the SONIC begins using the bus. The bus latency tolerance can be increased by programming the transmit FIFO threshold higher and the receive FIFO lower. The bus latency tolerance is calculated by the following equations:

   $$\text{TX FIFO Tolerance} = (\text{FIFO threshold}) * (0.8\ \mu\text{s})$$

   $$\text{RXFIFO Tolerance} = (32 - \text{FIFO threshold}) * (0.8\ \mu\text{s})$$

4) Do wait-states need to be added into the memory cycle?

   The SONIC can operate up to a 2 bus clock memory cycle. If this is too fast, you can program the SONIC to insert 1 to 3 wait-states for each memory cycle. A two clock memory cycle requires a memory access time of approximately 40 ns–50 ns. (Note that wait state can also be inserted by hardware using the $\overline{\text{RDYi}}$ or $\overline{\text{DSACK0}}$, $\overline{1}$ inputs.)

5) What type of packets do you want to accept?

   The SONIC is generally programmed to accept its own physical address and the Broadcast address. In some applications, however, the SONIC may be programmed to accept multiple physical/multicast addresses (up to 16), and errored and runt packets.

6) What is the maximum number of consecutive packets that you expect to receive?

   This question is perhaps the most difficult to answer since it deals with the upper level protocols. In many transport protocols, flow control is used by the receiving node to limit the number of consecutive packets the transmitting node may send unacknowledged. This is generally called the ''window size''. Ideally, the software provides the SONIC with the memory resources it needs to completely buffer a complete ''window''.

7) What types of interrupts do you want the system to respond to?

   The SONIC can generate a variety of interrupts. Not all interrupts, however, need be (or should be) used to generate interrupts to the system. For maximum performance, you want as few interrupts as possible. A typical system allows interrupts occurring from good receptions and transmissions, and errored transmissions.

Initialization Example

Once the above questions have been answered, you can begin coding the initialization routine. This routine has been divided into 9 steps, but, only steps 1 and 9 need to be followed in the order presented. Example code is provided in the appendix.

1) Reset the SONIC: When the SONIC is powered-on, the hardware generally resets the SONIC by pulsing the $\overline{\text{RESET}}$ pin low. Thus, software does nothing to reset the SONIC. Once reset, the SONIC remains in reset mode until the RST bit in the Command register is cleared. If the hardware does not provide the reset, the software can perform the functional equivalent by simply setting the RST bit. All initialization should be done in reset mode to prevent spurious actions by the SONIC.

2) Configure the System Interface: This step writes to the Data Configuration Register (DCR) to configure the SONIC's bus interface circuitry. The configuration information is found by answering questions 1 through 4, discussed above. Note that the DCR can only be written to in reset mode.

3) Set Up the Receive Filters: This step determines what types of packets to accept (i.e., Physical, Multicast, Broadcast, Runt, and Errored packets) and what addresses to accept. The type of packet to accept is programmed in the Receive Configuration register and the addresses to accept are programmed into the Content Address Memory (CAM). See section 2.2.4 for loading the CAM.

4) Enable the Interrupts: This step enables the interrupts by writing to the Interrupt Mask register (IMR). Note that the interrupting condition is indicated by the Interrupt Status Register (ISR), but will not generate an interrupt unless the corresponding IMR bit is set. Note also that if the SONIC is initialized in reset mode, no interrupts can be generated.

5) Initialize Memory: This step initializes the three memory structures used by the SONIC for transmission and reception and allocates the memory blocks for storing received packets. An initialization example is illustrated in *Figures 3-1* and *3-2*. The *non-shaded* areas indicate fields which must be initialized and *shaded* areas indicate fields which are written to by the SONIC.

There are a few caveats discussed below:

All Descriptor Areas:

- Descriptor must be aligned to word (16-bit) boundaries in 16-bit mode and aligned to double word (32-bit) boundaries in 32-bit mode.
- The Descriptor Areas must not cross over a 32k word boundary since it only operates within this range.
- In 32-bit mode, the upper 16 data bits, D<31:16> are not used.

Transmit Descriptor Area:

- The transmit buffers (Transmit Buffer Area) may be aligned to any boundary; that is, the TXpkt.ptr0, 1 fields may contain any value.
- The packet and fragment size may be as low as 1 byte; that is, the TXpkt.pkt__size and TXpkt.frag__size may contain the value of one.

Receive Resource Area

- The resource descriptors must be contiguous and can not straddle the endpoints.
- In the lower buffer pointer field, RXrsrc.ptr0, the SONIC ignores least significant bit in 16-bit mode and the 2 least significant bits in 32-bit mode. This forces receive buffers to always align to either word or double word boundaries.

6) Initialize the Buffer Management Registers: This step initializes the buffer management registers to the starting positions in the buffer management (see *Figures 3-1* and *3-2*). These initialized registers are shown in Table 3-1.

7) Issue RRA command: By setting the RRRA bit in the Command register, you force the SONIC to read the RRA. The SONIC reads the RRA beginning at the RRP location and loads the following registers. (For mnemonics description, see appendix.)

   CRBA0 ← RXrsrc.ptr0

   CRBA1 ← RXrsrc.ptr1

   RBWC0 ← RXrsrc.wc0

   RBWC1 ← RXrsrc.wc1

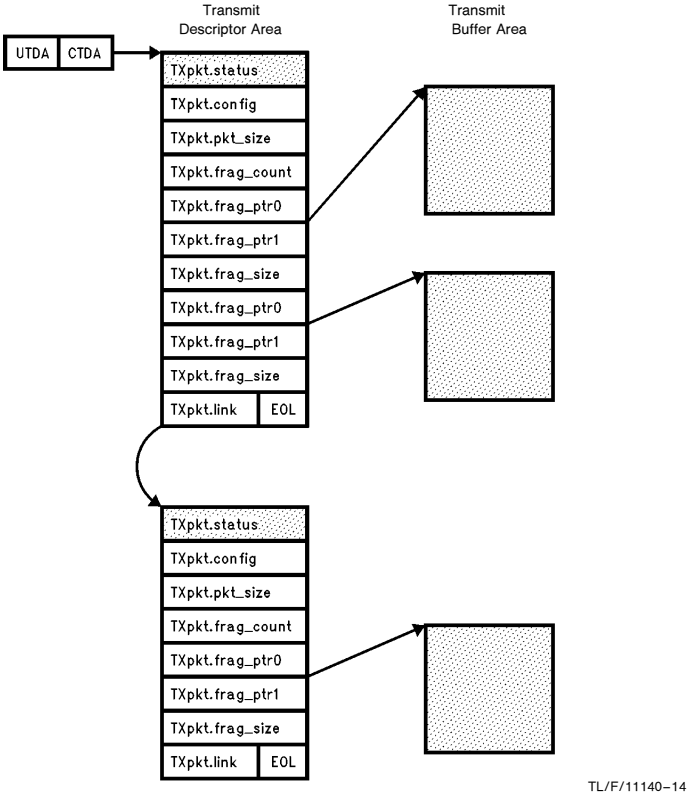
   After this command has executed (RRRA bit resets), the SONIC is ready to store the next packet in the first RBA allocated to it.

8) Clear and Tally Counters (optional): The tally counters (CRC, Frame Alignment, and Missed Packets) may be cleared by writing FFFFh to these registers. These counters will rollover after FFFFh is reached.

9) Bring the SONIC On-line: This last step commissions the SONIC to receive, transmit, and generate interrupts. The software enables the SONIC by setting the RXEN bit and clearing the RST bit in the Command register.

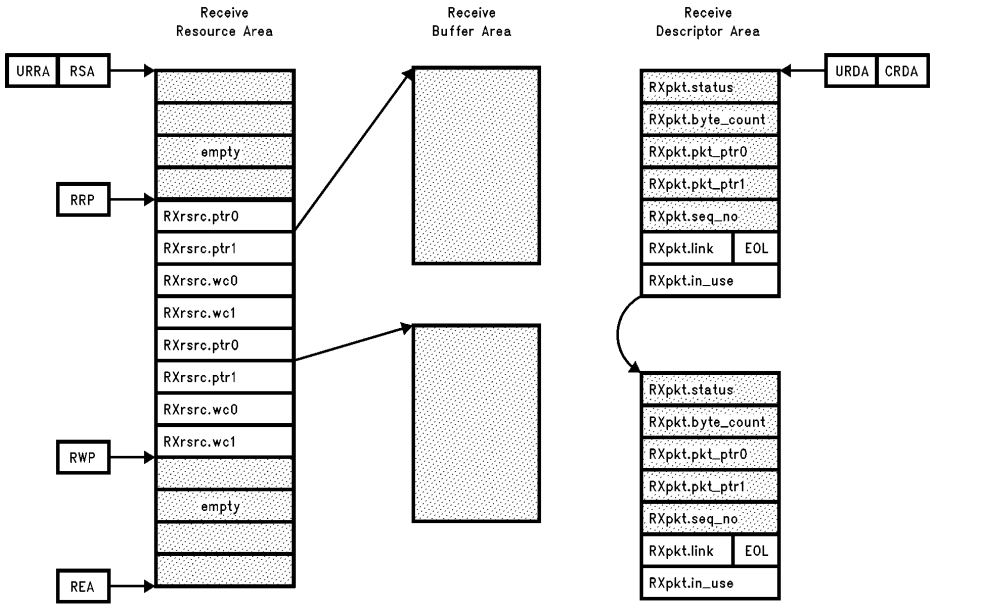
TABLE 3-1. Initialization of Buffer Management Registers

Reg.	Initialized with
URDA	A<31:16> of starting location of RDA
CRDA	A<15:1> of starting location of RDA
UTDA	A<31:16> of starting location of TDA
CTDA	A<15:1> of starting location of TDA
URRA	A<31:16> of starting location of RRA
RSA	A<15:1> of starting location of RRA
REA	A<15:1> of ending location of RRA
RRP	Points to first descriptor the SONIC reads
RWP	Points to next location where the software will place a descriptor



TL/F/11140–14

FIGURE 3-1. Initialization Example for Transmit Buffer Management (shaded areas not initialized)

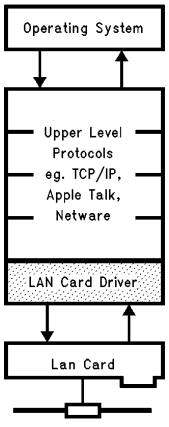


TL/F/11140–15

FIGURE 3-2. Initialization Example for Receive Buffer Management (shaded areas not initialized)

4.0 WRITING DRIVERS FOR THE SONIC

The Driver (see *Figure 4-1*), being the lowest level of software, shields the upper software levels from the details of the hardware. The Driver performs the required low-level transmit and receive functions such as passing packet up to the upper level software, recycling receive buffers, and enqueuing packets for transmission. The Driver performance is important since it may potentially receive packets at the full network rate. Any packet losses at this level can severely affect the overall performance of the network. This section describes the basic algorithms for writing a Driver for the SONIC. Example code is provided in the appendix.



TL/F/11140–16

FIGURE 4-1. Relationship of Driver of Upper Level Software

Overview

The Driver for the SONIC consist of two procedures, INITIATE__TX *(Figure 4-2)* and SONIC__ISR *(Figure 4-3)* for transmit and receive operations. During transmit operations, the upper level software first assembles packets for transmission by gathering the pointers to the fragments and then calling INITIATE__TX to begin the transmission. When the SONIC finishes transmission, it interrupts the system. The system then enters the interrupt service routine, SONIC__ISR, where it reports the status of the packets transmitted. During received operations, the SONIC also interrupts the system upon receiving a packet. The system enters SONIC__ISR to post status and then to pass the packet up to the upper level software via pointers.

4.1 INITIATE__TX

This procedure requires that all pointers to the fragments and the sizes of these fragments are passed down to it by the upper level software. It only initiates a packet for transmission; it does not report status. This action is performed by SONIC__ISR after the packet has been transmitted. INITIATE__TX operates as follows:

1) Obtains the pointers delivered by the upper level software and fills out a descriptor in the Transmit Descriptor area (TDA).
2) If the packet is less than 64 bytes, it pads it out to this length.
3) Issue the transmit command to the SONIC and return.

It is important that descriptors are appended in the manner prescribed in section 1.3. This algorithm improves performance by guaranteeing that the SONIC continues to transmit all packets in the descriptor list.

4.2 SONIC__ISR

This procedure is the interrupt service routine which responds to three interrupts generated by the SONIC: PACKET RECEIVED, TRANSMISSION DONE, and TRANSMIT ERROR. Interrupts occurring before and during the interrupt service routine are serviced before SONIC__ISR exits. SONIC__ISR is broken down into three main sections: (1) reading the cause of the interrupt, (2) processing received packets, and (3) posting status of transmitted packets. The first action performed is finding the cause of the interrupt. For receive interrupts, SONIC__ISR jumps to the receive routine, and for transmit interrupts (good and errored transmissions), it jumps to the transmit routine. The receive routine examines the first descriptor in the RDA, then passes the pointer of the packet up to the upper level software for further processing. It continues reading the RDA until it reaches the end of the descriptor list. The receive routine also recycles receive buffers as necessary. The transmit routine reads the first descriptor in the TDA and reports the status of the transmitted packet to the upper level software. If more than one packet has been enqueued, the transmit routine examines the complete list in the TDA. SONIC__ISR is summarized below.

Reading the Interrupt

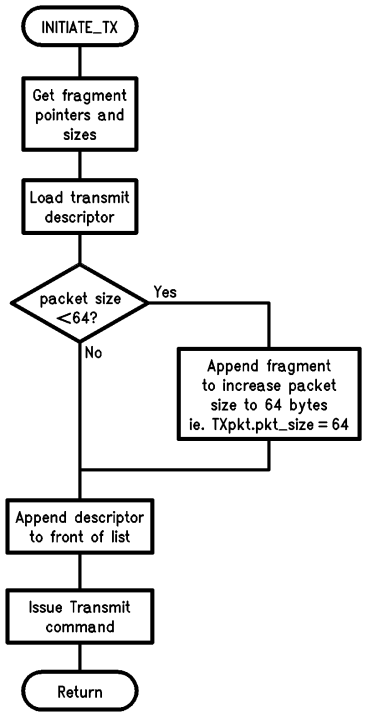
1) Read the Interrupt Status register for the cause of interrupt. If a transmit interrupt has occurred, go to step 2; if a receive interrupt has occurred, go to step 4; or if no more interrupts are present, return.

Transmit Routine

2) Read the next TXpkt.status in the Transmit Descriptor Area and post status to the upper level software.
3) Read the End of List (EOL) bit in the TXpkt.link field to determine if the current descriptor is the last descriptor. If it is not, go back to step 2 to post status of the remaining packets; otherwise go back to step 1.

Receive Routine

4) Read the next RXpkt.status field in the Receive Descriptor Area and pass the pointer and status of the packet up to the upper level software.
5) Read the RXpkt.seq__no field. If the RBA number is different from the previous one, enter the RBA number into the RBA ''scoreboard''. For more information, see section 1.2.
6) Check the LPKT bit from the RXpkt.status field. If set to ''1'', enter the packet sequence number (from the RXpkt.seq__no) into the RBA scoreboard.
7) Read the RXpkt.in__use field, if the field is cleared to all zeros, go back to step 4 to process the remaining packets; otherwise if RXpkt.in__use is not equal to zero, the end of the list has been reached; proceed to step 7.
8) Call the system to determine which packets have been processed by the upper level software. Tally the processed packets in he RBA scoreboard.
9) Find freed up RBAs and return them to the front of Receive Resource Area (RRA).
10) Find the freed up receive descriptors and return them to the front of the descriptor list; then go to step 1.



TL/F/11140–17

FIGURE 4-2. INITIATE__TX Routine

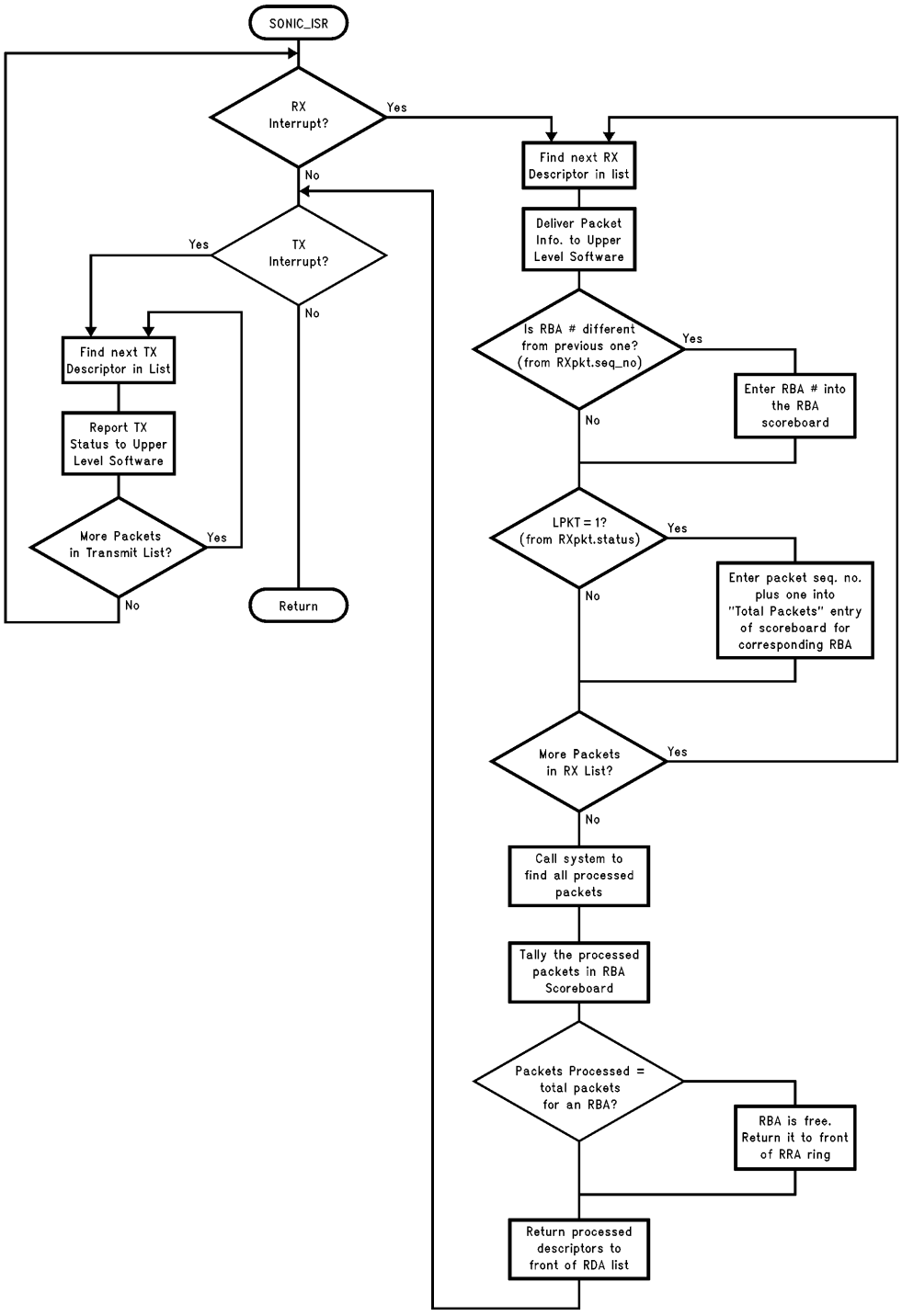


FIGURE 4-3. SONIC__ISR Routine

5.0 STRATEGIES FOR IMPROVING DRIVER PERFORMANCE

Making the Driver as efficient as possible is crucial for the overall performance of the network. Empirical results have shown that the difference between a poor and a good Driver can vary as much as 10% to 20%. The Driver is particularly vulnerable to becoming a bottleneck since it may, at times, be receiving data at the full network bandwidth (10 Mb/s). Any packets that are lost at the Driver level impacts all levels. While upper level protocols provide packet recovery mechanisms, these tend to be quite slow (on the order of seconds). Typically, software timers must time out before the upper level software retransmits an unacknowledged packet. In this section, some hints are discussed to make a fast Driver.

1) Write the Driver in assembly code: The fastest code is generally written in assembly code since people write more efficient code than a compiler. Writing your own assembly code also gives you the option to use some ''tricks'' which are not normally accepted as ''good'' programming practice. One such example is using a JUMP statement instead of a CALL statement. The JUMP statement, by nature, is quite messy, but is considerably faster since it involves less CPU cycles. Of course, the disadvantage in using assembly code is that it is less readable and portable. As a compromise, you may consider a good optimizing compiler.
2) Reduce the Number of Interrupts: Interrupts to the system inherently make it less efficient since the CPU must make a context switch between what it was currently doing to the interrupt service routine. This switch involves pushing the CPU registers onto the stack, jumping to an interrupt vector table, issuing an interrupt acknowledge to the interrupt controller, then executing the interrupt service routine. The overhead associated with each interrupt makes the CPU less efficient. The example interrupt service routine discussed in section 4.0, responded to interrupts generated from good transmission and receptions, and errored transmission. It is possible, however, to reduce the source of interrupts to just two, allowing only interrupts to occur from good receptions and errored transmissions. The reason good transmission interrupts may be eliminated is because the upper level software generally does nothing for these events. Only for an errored transmission must the upper level software intervene such as to retransmit the packet. Good transmissions, while they still need to be reported, can be status on a less timely basis such as after processing receive interrupts or after a specified time period. The SONIC's General Purpose timer can be used to generate such a time period.
3) Append Transmit Descriptors as described in section 1.0: The algorithm described guarantees that the SONIC continues to transmit all packets in the list, even if it has reached the point where the new descriptor(s) have been appended to the end of the list. If the algorithm is not followed, the SONIC may stop at the enjoining point and this forces the Driver to intervene.
4) Supply Sufficient Number of Receive Packet Descriptors: Since the receive descriptor uses a relatively small amount of memory (7 words or double words, depending on the data size mode), allocate sufficient number of them such that the SONIC never (or at least rarely) runs out of them. If the SONIC ever runs out of them, reception ceases, resulting in packet losses. The number of descriptors to allocate can be determined by answering question 6 of section 3.0.
5) Make the Receive Resource Area (RRA) Sufficiently Large: Since the RRA does not take up much memory (4 words or double words per descriptor), make it larger than the total number descriptors you expect to put into it. For example, if you expect you will need 10 resource descriptors, make the RRA large enough to accommodate 15 descriptors. Making the RRA larger than you will need, prevents the RRA from becoming a bottleneck in adding more resources.
6) Optimize the Size of the Receive Buffer Areas (RBAs): Generally speaking, the larger the RBAs, the more efficient the Driver. This is because the Driver handles fewer number of receive buffers and, thus, less processing time is dedicated to managing the buffers. There is a tradeoff, however. If the buffers are very large, the entire buffer areas are locked out for recycling so that large buffers become less space efficient in memory. As a guideline, 4k to 8k byte RBAs are good starting points for experimentation. Use larger buffers, if memory is plentiful.

6.0 SELF-TEST DIAGNOSTICS

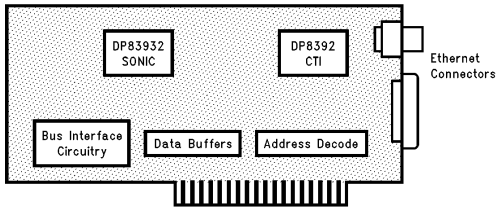
After the hardware has been designed and the Drivers written, there is still a need to verify that the hardware is still functioning. Rough shipping or improper handling (without static protection) can produce innumerable problems. Some boards which work fine in the lab invariably fail in the field. Thus, self-test diagnostics are used to determine the health of the boards and diagnose problems if something is amiss.

Figure 6-1 shows the basic components of the Ethernet system: address decode circuitry, data buffers, bus interface logic, Ethernet chipset (SONIC and transceiver) and the Ethernet connectors (BNC and 15-pin D). The Ethernet hardware can be fully tested by using the SONIC's three loopback modes. Each loopback mode is full-duplex, transmitting data as well as receiving it and are summarized below. An example routine is given in the appendix.

Mode 1: Data is routed back through the SONIC's MAC Unit. Both the transmit and receive Buffer Management operations are active and must be initialized accordingly. Verifies the MAC Unit, Bus interface logic, address decode circuitry and data buffers.

Mode 2: Similar to above, but data is routed back through the SONIC's ENDEC Unit. Verifies the SONIC's ENDEC unit.

Mode 3: Similar to above, but data is routed back at the transceiver. Verifies the Ethernet connectors (BNC and 15-pin D) and Ethernet transceiver (DP8392 CTI).



TL/F/11140–19

FIGURE 6-1. Basic Components of Ethernet Hardware

Appendix

A. Initialization Routine

```
/*****************************************************************/
/*                                                               */
/* Initialization Routine for SONIC                                  */
/*                                                               */
/*****************************************************************/

sonic_init ()
      {
      unsigned short init_RRA[512];   /* memory for RRA */

      /* initialize some registers */
      set_reg_value();

      /* allocate memory for TX descriptors and init UTDA and CTDA */
      init_tda()

     /* Init receive buffer area and RX registers  */
      Init_Des_page();
      Initial_RRA(RRA_NUM);
      Init_RDA(RDA_NUM);

      /* Issue Read RRA command */
      /* Must first bring SONIC out of reset before issuing any
         commands */
      REG_WRITE(card.crd_iobase+SONIC_cr*2, 0x0);
      REG_WRITE(card.crd_iobase+SONIC_cr*2, 0x0100);

      /* Bring SONIC on-line by enabling MAC receiver */
      REG_WRITE(card.crd_iobase+SONIC_cr*2, 0x0008);
      }
/*****************************************************************/
/* This routine initializes some of the SONIC's registers.     */
/*   ie., CR, DCR, RCR, IMR, ISR, CRCT, FAET, and MPT          */
/*****************************************************************/
set_reg_value()
{

      /* Put SONIC is reset */
      REG_WRITE(card.crd_iobase+SONIC_cr*2, 0x0080);

/* dcr value depends upon data width (16 or 32 bits) */
   #ifdef BIT32
      REG_WRITE(card.crd_iobase + SONIC_dcr*2, 0x00f9);
   #else
      REG_WRITE(card.crd_iobase + SONIC_dcr*2, 0x00d9);
   #endif

      REG_WRITE(card.crd_iobase + SONIC_rcr*2, 0x0000);
      REG_WRITE(card.crd_iobase + SONIC_imr*2, 0x3fff);
   /* Clear ISR  */
      REG_WRITE(card.crd_iobase + SONIC_isr*2, 0xffff);
   /* Clear Tally counters by writing FFFFh to them */
      REG_WRITE(card.crd_iobase + SONIC_crct*2, 0xffff);
      REG_WRITE(card.crd_iobase + SONIC_faet*2, 0xffff);
      REG_WRITE(card.crd_iobase + SONIC_mpt*2, 0xffff);
}
```

TL/F/11140–20

```
/*************************************************************
 *
 * Allocate memory for TDA and initialize UTDA and CTDA registers.
 *
 *************************************************************/

init_tda()
{
      short i;
      unsigned long addr;
      unsigned long tda1_start, tda2_start, tda3_start;
      unsigned short saddr;
      unsigned short u16, l16;

      /* Allocate memory for TDAs */
      tda1=(ONE_FRAG_TDA *) malloc(sizeof(ONE_FRAG_TDA) + 2);
      tda1_start = (unsigned long) tda1;
      tda2=(TWO_FRAG_TDA *) malloc(sizeof(TWO_FRAG_TDA) + 2);
      tda2_start = (unsigned long) tda2;
      tda3=(TWO_FRAG_TDA *) malloc(sizeof(TWO_FRAG_TDA) + 2);
      tda3_start = (unsigned long) tda3;

      /* Force TX descriptors to double word alignment */
   #ifdef BIT32
      if ( (tda1_start & 0x00000003) == 0)
            ;
      else
            tda1_start += 2;
      if ( (tda2_start & 0x00000003) == 0)
            ;
      else
            tda2_start += 2;
      if ( (tda3_start & 0x00000003) == 0)
            ;
      else
            tda3_start += 2;
   #endif
      /* Convert the double word alignment address to pointer */
      tda1=(ONE_FRAG_TDA *) tda1_start;
      tda2=(TWO_FRAG_TDA *) tda2_start;
      tda3=(TWO_FRAG_TDA *) tda3_start;

      /* Finding effective address of TDA1 to load UTDA and CTDA regs.*/
      addr=(unsigned long) tda1;    /* Using large mem. model..*/
                  /* addr is the address in 8086 format */
                  /* upper 16 bits = BASE, lower 16 bits = OFFSET */
      u16 = addr >> 16;
      l16 = addr;
      addr=(unsigned long) u16 * 16 + l16;
      u16 =addr >> 16;
      REG_WRITE(card.crd_iobase+SONIC_utda*2, u16);
      REG_WRITE(card.crd_iobase+SONIC_ctda*2, addr);
      }
```

TL/F/11140–21

```
/*****************************************************************
      Name
            Initialize Descriptor Page
      Syntax
            Init_Des_Page();
      Description
            This function  gets 3 4K consecutive bytes of memory
            from  the host for the RBA. Also initializes the URRA
            and CDP registers.

      Input
            None.

      Author
            Michael Lui

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

short Init_Des_Page()

{
      unsigned short urra;    /* upper 16 bits of the beginning
                                    addr of RRA */
      unsigned short urda;    /* upper 16 bits of the beginning
                                 addr of RDA */
      unsigned short cdp;           /* beginning address of the cdp */
      R_DESCRIPTOR *temp_RDA;
      unsigned short i;             /* index */
      unsigned long laddr;
      unsigned long addr;
      long EA();
      unsigned long rba1_start, rba2_start, rba3_start;
      unsigned short u16, l16;

     /* allocate memory to RDAs */
      L_RDA=F_RDA=NULL;
      for (i=0; i<RDA_NUM; i++) {
      temp_RDA=(R_DESCRIPTOR *) malloc(sizeof(R_DESCRIPTOR) + 2);

      /* force double word alignment for RX descriptor */
#ifdef BIT32
      addr = (unsigned long) temp_RDA;
      if ((addr & 0x00000003) == 0)
            ;
      else
            addr += 2;
      temp_RDA = (R_DESCRIPTOR *) addr;
#endif
      temp_RDA->next=NULL;
      if (F_RDA == NULL)
            L_RDA=F_RDA=temp_RDA;
      else {
            L_RDA->next=temp_RDA;
            L_RDA=temp_RDA;
      }
      }
      /* allocate memory for RBA */
      init_RBA1=(unsigned char *) malloc(4100);
```

TL/F/11140–22

```
        init_RBA2=(unsigned char *) malloc(4100);
        init_RBA3=(unsigned char *) malloc(4100);

        rba1_start=(unsigned long) init_RBA1;
        rba2_start=(unsigned long) init_RBA2;
        rba3_start=(unsigned long) init_RBA3;

        /* forcing double word alignment for RBAs.  */
ifdef BIT32
        if ( (rba1_start & 0x00000003) == 0)
        ;
        else
        rba1_start+=2;
        if ( (rba2_start & 0x00000003) == 0)
        ;
        else
        rba2_start+=2;
        if ( (rba3_start & 0x00000003) == 0)
        ;
        else
        rba3_start+=2;
#endif
        /* Convert double word alignment address to pointer */
        RBA1 = (unsigned char*) rba1_start;
        RBA2 = (unsigned char*) rba2_start;
        RBA3 = (unsigned char*) rba3_start;

        /* initialize  URRA and CDP registers */
        RRA_start = (unsigned long) init_RRA;
        /* check RRA is aligned on double word boundary */
ifdef BIT32
        if ( (RRA_start & 0x00000003) == 0)
        ;
        else
        RRA_start+=2;
#endif

        /* Assign urra */
        laddr = (unsigned long) RRA_start;
        u16=laddr >> 16;
        l16=laddr;
        laddr = (unsigned long) u16 * 16 + l16;
        urra = laddr >> 16;
        /* Load the URRA register */
        REG_WRITE(card.crd_iobase+SONIC_urra*2, urra);

        /* load the CDA descriptor pointer */
        laddr = (unsigned long)u16 * 16 +l16 +CAM_OFFSET;
        cdp=laddr;
        /* load the CDP register */
        REG_WRITE(card.crd_iobase+SONIC_cdp*2,cdp);
}
```

TL/F/11140–23

```
/*************************************************************
      Name
      Initialize RRA

      Syntax
      flag=Init_RRA(n);

      Description
      This function will create a circular queue with n
      number of RRA descriptors in it. The RRA descriptors
      are pointing to the     corresponding RBA blocks. It will
      also load the RSA, REA, RRP, and RWP registers.

      Returned Value
      1 = Success
      0 = Failed

      Author
      Michael Lui

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

short Initial_RRA()
{
      struct sonicreg *sonic=0;
unsigned short rsa;           /* Resource Start Area */
unsigned short rea;           /* Resource End Area */
unsigned short rrp;           /* Resource Read Pointer */
unsigned short rwp;     /* Resource Write Pointer */
unsigned short urba;    /* Upper 16 bit of the RBA starting
                                  address */
unsigned short lrba;    /* Lower 16 bit of the RBA starting
                                  address */
unsigned short i;       /* for loop index */
      unsigned short low_addr;
      unsigned short high_addr;
      unsigned long addr,laddr;
short inc;              /* RRA increment */
     unsigned short u16, l16;

      addr = (unsigned long) RRA_start;
      u16=addr >> 16;
      l16=addr;
addr = (unsigned long) u16 * 16 + l16;

      /* Lower 16 bit of the RRA */
rsa = (unsigned short) addr;
      /* Load the RSA Register */
      REG_WRITE(card.crd_iobase+SONIC_rsa*2, rsa);

laddr=addr + RWP_OFFSET;
rea = (unsigned short) laddr; /* Ending address of RRA */
      /* Load the REA Register */
      REG_WRITE(card.crd_iobase+SONIC_rea*2, rea);

rrp = rsa;  /* Read Pointer starts at the beginning
                        address */

      /* Load the RRP Register */
```

TL/F/11140–24

```
      REG_WRITE(card.crd_iobase+SONIC_rrp*2, rrp);

laddr = addr + RWP_OFFSET/2;
rwp = (unsigned short) laddr; /* Only 3 descriptors
                                        initially */
      /* Load the RWP Register */
      REG_WRITE(card.crd_iobase+SONIC_rwp*2, rwp);

      /* Initialize the RRA descriptors */
      RRA=RRA_start;
      /* for 32-bit memory each descriptor uses a double word, for
         16-bit memory, each descr. uses a word. */
#ifdef BIT32
      inc=4;
#else
      inc=2;
#endif
      /* Load RBA1 address */
     addr=(unsigned long) RBA1;
     u16=addr >> 16;
     l16=addr;
      addr=(unsigned long)u16 * 16 + l16;
      low_addr = addr & 0x0000ffff;
      * (unsigned long *)RRA = low_addr;
      RRA +=inc;
      * (unsigned long *)RRA = addr >> 16;
      RRA +=inc;
      /* Load RXrsrc.buff_wc0 */
      * (unsigned short *)RRA = 0x0800;
      RRA +=inc;
      /* Load RXrsrc.buff_wc1 */
      * (unsigned short *)RRA = 0;
      RRA +=inc;

      /* Load RBA2 address */
     addr=(unsigned long) RBA2;
     u16=addr >> 16;
     l16=addr;
      addr=(unsigned long) u16 * 16 + l16;
      low_addr = addr & 0x0000ffff;
      * (unsigned short *)RRA = low_addr;
      RRA +=inc;
      * (unsigned short *)RRA = addr >> 16;
      RRA +=inc;
      /* Load RXrsrc.buff_wc0 */
      * (unsigned short *)RRA = 0x0800;
      RRA+=inc;
      /* Load RXrsrc.buff_wc1 */
      * (unsigned short *)RRA = 0;
      RRA +=inc;

      /* Load RBA3 address */
     addr=(unsigned long) RBA3;
     u16=addr >> 16;
     l16=addr;
      addr=(unsigned long)u16 * 16 + l16;
      low_addr = addr & 0x0000ffff;
      *(unsigned short *)RRA = low_addr;
      RRA+=inc;
      * (unsigned short *)RRA = addr >> 16;
```

TL/F/11140–25

```
        RRA+=inc;
        /* Load RXrsrc.buff_wc0 */
        * (unsigned short *)RRA = 0x0800;
        RRA+=inc;
        /* Load RXrsrc.buff_wc1 */
        * (unsigned short *)RRA = 0;
          RRA+=inc;
}
```

TL/F/11140–26

```
/*************************************************************
      Name
      Initialize RDA

      Syntax
      flag = Init_RDA(n);

      Description
      This function will create a linked list of some
      arbitrary number of packet descriptors. The EOL bit for
      the last    descriptor should set to 1 while the others
      should set to 0.  The in_use field should set to a
      non-zero value for all descriptors. The CRDA register
      should loaded with the address of the first descriptor.

      Returned Value
      1 = Success
      0 = Failed

*************************************************************/
short Init_RDA()
{
unsigned long crda;     /* Current CRDA Register */
      unsigned char *RDA; /* RDA address */
      R_DESCRIPTOR *cur_RDA;  /* current RDA */
unsigned short n_RDA_addr;    /* next RDA address */
      unsigned long addr;
      short i;
      unsigned u16, l16;

crda = (unsigned long)  F_RDA;
      u16 = crda >> 16;
      l16 = crda;
      crda = (unsigned long)u16 * 16 + l16;

      /* Load the CRDA Register */
REG_WRITE(card.crd_iobase+SONIC_crda*2, crda);

cur_RDA=F_RDA;
      while (cur_RDA->next != NULL)
      {
          addr = (unsigned long) cur_RDA->next;
          u16 = addr >> 16;
          l16 = addr;
      addr=(unsigned long) u16 * 16 + l16;
      n_RDA_addr=(unsigned short) addr;
      cur_RDA->pkt_link=n_RDA_addr;
          cur_RDA->status=0;
          cur_RDA->byte_count=0;
          cur_RDA->pkt_ptr0=0;
          cur_RDA->pkt_ptr1=0;
          cur_RDA->seq_no=0;
      cur_RDA->in_use=0xffff;
      cur_RDA=cur_RDA->next;
      }
      /* last descriptor */
cur_RDA->pkt_link=0x0001;     /* last desr. has EOL = 1 */
      cur_RDA->in_use=0xffff;
      lrda = cur_RDA;
}
```

TL/F/11140–27

B. Initiate Transmission Routine

```
/*********************************************************
*
Driver_send().  This routine, called by the upper level
software, gets the byte count, pointers to fragments and
the fragment sizes, enters these parameters into the TDA,
then initiates a transmission.

*********************************************************/
driver_send(ptr)
pktstruc *ptr     /*pointer to structure which gives
                    pkt_size, frag_count, frag_size */
{

    /* Fill out TDA */
    tda->pkt_size=packet_size;
    tda->frag_count=fragment_count;
    for (i=0; i<fragment_count; i++)
        Fill_fragment_ptr_size();

    /* Check packet length; if less than 46 bytes, add pad */
    Check_pkt_length();


    /* Get address of next TX descriptor to use */
    tda->link = get_next(); /* returns addr. of descr. */
    /* Set EOL.to 1.  */
    tda->link |= 0x1;


    /* ISR will Set this flag to 1 */
    xmit_interrupt=0;

    /* Issue transmit command */
    REG_WRITE(card.crd_iobase+SONIC_cr*2, CMD_TXP);


}
```

TL/F/11140–28

C. Interrupt Service Routine

```
/*****************************************
 Interrupt Service Routine

 (For simplicity the code for recycling RBAs
 has heen removed.)
 *****************************************/

interrupt _sonic_isr()
{
    unsigned short imr, isr, mask;
    unsigned int status, byte_count;
    int oldinterrupts; long temp_ptr, ptr;

    mask=0;
    /* mask the imr */
    REG_WRITE(card.crd_iobase+SONIC_imr*2, mask);

    while (isr=REG_READ(card.crd_iobase+SONIC_isr*2)) {
        if (isr & ISR_PKTRX) {
            /* reset PKTRX bit */
            REG_WRITE(card.crd_iobase+SONIC_isr*2,ISR_PKTRX);

            /* Process receive packets */

            while (cur_rda->in_use == 0) {
                    TotalRxPacketCount++;
                    status = cur_rda->status;
                    byte_count = cur_rda->byte_count;
                    temp_ptr = cur_rda->ptr1;
                    temp_ptr = temp_ptr<<16;
                    ptr = temp_ptr | cur_rda->ptr0;
                    /* Report packet to upper level software */
                    packet_received(status,byte_count,ptr);

                    /* Processing packets in order, when LPKT is 1,
                       update the RWP register */
                    if (cur_rda->status==RCR_LPKT) {
                            cur_rwp=cur_rwp->next;
                            /* advance rwp */
                            REG_WRITE(card.crd_iobase+SONIC_rwp*2,
                                                cur_rwp->loc);
                    }

                    /* finish up receive */
                    if (cur_rda->in_use == 0) {
                        cur_rda->in_use=0x0ffff;
                        cur_rda->pkt_link |= 0x1;
                        lrda->pkt_link &= 0x0fffffffe;
                        lrda=cur_rda;
                        cur_rda=cur_rda->next;
                    }
            }

            /* check for RBE overflow (required) */
             isr=REG_READ(card.crd_iobase+SONIC_isr*2);
             if (isr & ISR_RBE) {
                   /* Increment buffer overflow counter */
```

TL/F/11140–29

```
                RRAExhaustCount++;
                /* reset RBE, this also causes the SONIC to read
                   the RRA */
                REG_WRITE(card.crd_iobase+SONIC_isr*2,R_RBE);
            }

            /* check for RDE overflow (optional) */
            if (isr & ISR_RDE) {
                RDAExhaustCount++;
                REG_WRITE(card.crd_iobase+SONIC_isr*2, ISR_RDE);
            }
        }

        /* Process transmitted packets */
        else if (isr & (ISR_TXER|ISR_TXND)) {
            xmit_interrupt=1;
            REG_WRITE(card.crd_iobase+SONIC_isr*2, ISR_TXER|ISR_TXDN);
            while (1) {
                if (tda->status & TCR_PTX) {/* Successful TX occurred */
                    TotalTxPacketCount++;

                    /* Post status of transmitted packet to
                       upper level software */
                    packet_tx(TX_status);
                  /* Increment counters for net. management. */
                    if (tda->status & TCR_DEF)
                        DeferXmissionCount++;
                    if (tda->status & TCR_NCRS)
                        NoCRSCount++;
                    if (tda->status & TCR_CRSL)
                        CRSLostCount++;
                    if (tda->status & TCR_OWC)
                        OutOfWindowCollisionCount++;
                    if (tda->status & TCR_PMB)
                        PacketMonitorBadCount++;
                }
                /* TX abort condition occured. CTDA register points to
                   last descriptor attempted. */
                else {
                    /* Increment counters for net. management. */
                    if (tda->status & TCR_EXD)
                        ExcessDeferalCount++;
                    if (tda->status & TCR_EXC)
                        ExcessCollisionsCount++;
                    if (tda->status & TCR_FU)
                        FIFOUnderRunCount++;
                    if (tda->status & TCR_BCM) {
                        ByteCountMismatchCount++;
                        tda->pkt_size=Total_fragment_size(tda);
                    }
                    if (--RetryCounter == 0)
                        HardTransmitErrorCount++;

                        /* Post status of transmitted packet to
                           upper level software that packet was undeliverable
*/
                           packet_tx(TX_status);
                    else {
                        /* resend the same packet again up to RetryCounter */
                        REG_WRITE(card.crd_iobase+SONIC_cr*2, CMD_TXP);
```

TL/F/11140–30

```
                    }
                }
                /* look for last descriptor in TX list */
                if (tda->link & 0x1)
                    break;
                else
                    tda=tda->next;
            }
        }
    }
    pic_eoi(card.crd_interrupt);
    REG_WRITE(card.crd_iobase+SONIC_imr*2, card.crd_intmask);
}
```

TL/F/11140–31

D. Diagnostic Routine

```
sonic_diag ()
      {
      int oldinterrupt;
      struct aclock *clk, *alock_alarm();
      long timeout=0;
      unsigned short temp, u16, l16, addr;
      unsigned long laddr;
        extern int timeout_func();
        short result;

      /* Before loopback test can commence SONIC needs to be
         initialized */

        /* check BNC cable connection:  If transmission does not
           finish after specified time period (~1sec), the BNC
           connector is not connected.  If excessive collisions
           occur, the cable is not terminated */

      clk=aclock_alarm(50,50,timeout_func, &timeout);
        REG_WRITE(card.crd_iobase+SONIC_isr*2, 0xffff);
        /* Get the 1st tda */
        laddr=(unsigned long) tda1;
        u16=laddr >> 16;
        l16=laddr;
        laddr=(unsigned long)u16 * 16 + l16;
        addr=(unsigned short) laddr;
        REG_WRITE(card.crd_iobase+SONIC_ctda*2, addr);
        tda1->link=0x0001;

      /* Issue transmit command */
      REG_WRITE(card.crd_iobase+SONIC_cr*2, CMD_TXP);
      for (timeout_value=0; timeout_value < 2; ) {
              temp=REG_READ(card.crd_iobase+SONIC_isr*2);
                if (temp & (ISR_TXDN | ISR_TXER))
                    break;
      }
      clock_kill(clk);
      if (timeout_value) {
          check_cable=2;      /*Timeout occurred, BNC not connected*/
          goto final;
      }
      else if (tda1->status & TCR_EXC) {
              check_cable = 3;
              goto final;     /* Exc. Coll. occurred, cable not
                                 terminated */
      }
      else
              check_cable = 1;

        /* MAC loopback */
      laddr = (unsigned long) F_RDA;
      u16=laddr >> 16;
      l16=laddr;
      laddr = (unsigned long)u16 * 16 + l16;
      addr = (unsigned short) laddr;
      REG_WRITE(card.crd_iobase+SONIC_crda*2, addr);
        mac_loopback=loopback(0x0200);
```

TL/F/11140–32

```
        if (mac_loopback != 1)
            goto final;

        /* ENDEC loopback */
      laddr = (unsigned long) F_RDA;
      u16=laddr >> 16;
      l16=laddr;
      laddr = (unsigned long)u16 * 16 + l16;
      addr = (unsigned short) laddr;
      REG_WRITE(card.crd_iobase+SONIC_crda*2, addr);
        endec_loopback=loopback(0x0400);
        if (endec_loopback != 1)
            goto final;

        /* transceiver loopback */
      laddr = (unsigned long) F_RDA;
      u16=laddr >> 16;
      l16=laddr;
      laddr = (unsigned long)u16 * 16 + l16;
      addr = (unsigned short) laddr;
      REG_WRITE(card.crd_iobase+SONIC_crda*2, addr);
        trans_loopback=loopback(0x0600);
        if (trans_loopback != 1)
            goto final;

      return(ok);

final:
      return(error);    /* one of the loopback test failed/*
}


/* This routine is to perform the loopback tests */

loopback( rcr_mode)
unsigned short  rcr_mode;
{
        struct aclock *clk;
        unsigned short temp, u16, l16, addr, rcr_value;
        unsigned long laddr;
        long timeout=0;
        short i;
        struct aphys *phys;

        /* Set up the clock to measure timeout */
      clk=aclock_alarm(50,50,timeout_func, &timeout);
        REG_WRITE(card.crd_iobase+SONIC_isr*2, 0xffff);
        /* Get the 1st tda */
        laddr=(unsigned long) tda1;
        u16=laddr >> 16;
        l16=laddr;
        laddr=(unsigned long)u16 * 16 + l16;
        addr=(unsigned short) laddr;
        tda1->link=0x0001;
        rcr_value=rcr_mode|0x3800;
        REG_WRITE(card.crd_iobase+SONIC_rcr*2, rcr_value);

        /* Out of reset mode */
        REG_WRITE(card.crd_iobase+SONIC_cr*2, 0);
```

TL/F/11140–33

```
        REG_WRITE(card.crd_iobase+SONIC_ctda*2, addr);
        REG_WRITE(card.crd_iobase+SONIC_cr*2, CMD_RXEN);
        /* Issue transmit command */
        REG_WRITE(card.crd_iobase+SONIC_cr*2, CMD_TXP);
      for (timeout_value=0; timeout_value < 2; ) {
             temp=REG_READ(card.crd_iobase+SONIC_isr*2);
               if (temp & (ISR_TXDN | ISR_TXER))
                   break;
      }
      clock_kill(clk);
      if (timeout_value)
          return(2);                /* timeout error */
      else if (tda1->status & TCR_PTX) {
               if (F_RDA->status & RCR_LBK) {
                   F_RDA->in_use=0xffff;
                   return(1);       /* good TX and RX status */
               }                    /*   loopback OK */
               else {
                   F_RDA->in_use=0xffff;
                   return(3);       /* Bad RX status error */
               }
        }
        else {
            F_RDA->in_use=0xffff;
            return(4);              /* Bad TX status error */
        }
}
```

TL/F/11140–34

National Semiconductor Corporation
2900 Semiconductor Drive
P.O. Box 58090
Santa Clara, CA 95052-8090
Tel: 1(800) 272-9959
TWX: (910) 339-9240

National Semiconductor GmbH
Livry-Gargan-Str. 10
D-82256 Fürstenfeldbruck
Germany
Tel: (81-41) 35-0
Telex: 527649
Fax: (81-41) 35-1

National Semiconductor Japan Ltd.
Sumitomo Chemical
Engineering Center
Bldg. 7F
1-7-1, Nakase, Mihama-Ku
Chiba-City,
Ciba Prefecture 261
Tel: (043) 299-2300
Fax: (043) 299-2500

National Semiconductor Hong Kong Ltd.
13th Floor, Straight Block,
Ocean Centre, 5 Canton Rd.
Tsimshatsui, Kowloon
Hong Kong
Tel: (852) 2737-1600
Fax: (852) 2736-9960

National Semiconductores Do Brazil Ltda.
Rue Deputado Lacorda Franco
120-3A
Sao Paulo-SP
Brazil 05418-000
Tel: (55-11) 212-5066
Telex: 391-1131931 NSBR BR
Fax: (55-11) 212-1181

National Semiconductor (Australia) Pty, Ltd.
Building 16
Business Park Drive
Monash Business Park
Nottinghill, Melbourne
Victoria 3168 Australia
Tel: (3) 558-9999
Fax: (3) 558-9998