APPLICATION NOTE

AN2451

Software workarounds for Philips PDI1394L11 Errata

Elie Boujaoude

1998 Aug 10

Philips
Semiconductors

PHILIPS

Software workarounds for Philips PDI1394L11 Errata

AN2451

Prepared by: Elie Boujaoude, Philips Semiconductors, Albuquerque, New Mexico

CONTENTS

1. INTRODUCTION

The PDI1394L11 is a Link Layer Controller chip that complies with IEEE 1394-1995. As in any development, some implementation errors and bugs were discovered after the chip has been released. The Errata sheet titled “PHILIPS ERRATA TO THE PDI1394L11 1394 AV LINK LAYER CONTROLLER” describes these errors by listing their expected behavior, their observed behavior (as implemented), and devises a short description for a workaround.

The scope of this application note is to devise software implementation of the workarounds for the errors listed in the above errata sheet when such a workaround is possible. Furthermore, these workarounds apply up to version 2 of the PDI1394L11 silicon. This version is saved in the ID register (IDREG address 0x000) of the Link control registers as described in the datasheet. The code presented for the workarounds is tailored to work with up to ver 1.2 of the Philips Semiconductors L11/P11 RDK software. That is due to the fact that the PDI1394L11 Link Layer is being used in various applications using different platforms, and that most designers and implementers of these applications have used the RDK software as the cornerstone for their software implementation. Also, the code for the workarounds uses functions that are defined in the RDK software. The new and modified code is presented in ***italic bold*** while the existing code is presented in *italic*. Finally, the applications implementers are expected to select the workarounds that pertain to their applications and port them into their environment.

In case of any questions or suggestions regarding these workarounds or any other related 1394 issues, please feel free to contact the Philips 1394 Applications group at 1394@abq.sc.philips.com

2. SOFTWARE IMPLEMENTATIONS OF THE WORKAROUNDS

2.1 E-1: Receive and Filter Only RESPONSE Packets, ARXALL

Description of expected operation: The ARXALL bit is designed to filter out unsolicited response packets. When set (1), all responses are stored. When cleared (0) only solicited responses are stored.

Description of observed behavior: When set (1) it works as expected. When cleared (0) AND the node is waiting for a response, only solicited responses are stored. When cleared (0) AND the node is NOT waiting for a response, all responses are stored, including unsolicited responses.

Solution or workaround: No workaround.

Software implementation: No software workaround.

2.2 E-2: Cycle Time Out, CYTMOUT, Occasionally Set Incorrectly After Bus Reset

Description of expected operation: The Link generates cycle timeout indications during bus reset. This signal indicates that the time between cycle start packet and the first subsequent subaction gap exceeded 125 microseconds.

Description of observed behavior: If the AV transmitter or AV receiver is active during a bus reset, a cycle time out (CYTMOUT) interrupt will generally gets set. If this node is cycle master then it will no longer send cycle start packets.

Solution or workaround: Cycle master functionality can be restored using the following procedure: clear CYMASTER (0) (bit 11 of LNKCTL register), acknowledge CYTMOUT (1), then reset CYMASTER (1).

Software Implementation: This workaround can be implemented in the RDK as follows:

After a bus reset, the Link is initialized by calling function “InitLink()” which is located in “Main.c”. This function currently clears the cycle master, and if the current node is root, the cycle master is restarted. Thus, the cycle timeout bit can be acknowledged just before the cycle master is restarted. Note that there is no need to check if the cycle timeout has been asserted before acknowledging it.

Main.c

```
void
InitLink(void)
{
    /* If node is root, set cycle master bit */
    CLR_CYMASTER();

    /*
        Errata E-2 Workaround.
        E-2: Cycle Time Out, CYTMOUT, Occasionally Set Incorrectly After Bus Reset
        Workaround: If the AV transmitter or AV receiver is active, and the cycle timeout
        gets asserted, then acknowledge the later before restarting the cycle master:
    */

    AVLinkWrite(0x00000020, LNKPHYINTACK);

    if (ROOT())
        SET_CYMASTER();

    /* Start timer */
        SET_CYTMREN();

    /* Configure to reject packets sent outside isoch. cycles */
        SET_STRCTISOC();

    /* Clear Asynch. reset Tx bit */
        CLR_ATXRST();

    /* Set Bus ID to 3FF */
        RESET_BUS_ID();

    ClrInterrupts();

    /* Init isoch. and asynch. communications */
    Init1394Com();

}
```

2.3 E-3: Confirmation Packet Not Received After Broadcast Packet is Sent

Description of expected operation: When an asynchronous broadcast packet is transmitted, a synthesized confirmation quadlet should appear in the Receive Request Queue, and the Read Request Quadlet Available (RREQQQAV) interrupt would be set.

Description of observed behavior: The confirmation packet will be stored in the link layer controller but it will not be available in the Read Request Queue. Several of these confirmation packets can accumulate. These packets will become available in the Read Request Queue as soon as a non-broadcast packet is received.

Solution or work around: If you receive confirmation packets unexpectedly, disregard them.

Software Implementation: This workaround is illustrated in the chat mode function that is provided by the Monitor program of the RDK. The chat mode provides the user the ability to send messages between two or more nodes. This function is implemented by sending the messages as asynchronous broadcast packets.

Using two connected nodes, node 1 and node 2, send several consecutive messages from node 1. The expected behavior is to receive one confirmation packet after each sent message. Although the confirmation packets are stored in the link layer controller, they are not available at the Read Request Queue due to the above listed bug (E-3). Once a non-broadcast packet is sent by node 1, or a packet is received, the confirmation packets are then available at the request queue. Thus, if a message is sent from node 2, then node 1 would get a RREQQQAV indicating that there is something in the read request queue. The later queue would contain the confirmation packets for the sent messages followed by the message sent from node 2. In such a case, the software needs to discard the confirmation packets in order to retrieve the message sent by node 2. This is illustrated in function “AsyncChat()” in module “Asytst.c” of the monitor program as follows:

Asytst.c - AsyncChat()

```
void
AsyncChat(void)
{
        …

   while (1) /* Stay here until the user types ”quit” */
   {
        /* look for a message */
        if (AsyncInside(1, &byte))
        {
           …
           switch (byte)
           {
                case REQ_RECEIVE:
                   if (!AsyncInside(1, message))
                   {
                        …
                   }
                   if (message[0] == BLOCK_WRITE_REQUEST)
                   {
                        …
                   }
                   else if(message[0] == CONFIRMATION_PACKET)
                   {
                        if (!AsyncInside(6, message+1))
                        {
```

```
                        printf("Error, exit this menu\n");
                        break;
                    }
                    if (message[1] != 0xFF )    /* verify that it is */
                                                /* a message       */
                    {
                        printf("Error\n");
                            break;
                    }
/*                  printf("  confirmed<<\n");*/
                }
                break;

            …

            default:
                …
                break;

        }       /* end of switch */
    }       /* end of if()   */

    …

    /* continue the while(1) */
  }
}
```

Note that the confirmation packets are read off the queue, and then discarded. The line

```
/*                  printf("  confirmed<<\n");*/
```

is intentionally commented and left as part of the code as a marker for the developer of the location where the confirmation messages would be displayed to the user if it was not for the bug.

2.4 E-4: SIDQAV is not always Set Correctly

Description of expected operation: SIDQAV is set upon receipt of a self-ID packet.

Description of observed behavior: Depending upon precisely when a bus reset is seen, occasionally the SIDQAV interrupt bit may not be set.

Solution or workaround: Use the bus reset interrupt bit along with the RREQQQAV interrupt to determine when self ID quadlets are in the queue.

See also the workaround for E-5.

Software Implementation: This workaround is illustrated in the RDK in function “ReadCheckSelfID()” in module “Selfid.c”. This function is called right after a bus reset to collect and check the self-ID packets. At entry of the function, there is a check to see if either SIDQAV bit or RREQQQAV bit is set. This check is illustrated in function “SIDAQV()” that is defined in module “driver.h”. If the check is successful, then ReadCheckSelfID() proceed in reading and checking the self-ID quadlets. Otherwise, a return value of FALSE is returned.

Driver.h

```
…
#define SIDQAV()        (LinkRead(ASYINTACK) & 0x4100)
…
```

SelfID.c - ReadCheckSelfID()

```
BOOLEAN
ReadCheckSelfID(void)
{
    …
    /* Look if self–ID packets are present */
    if (!SIDQAV())
    {
        selfIDProcessResultFlag = 1;
        return FALSE;            /* No selfID available */
    }
    …
}
```

Refer to the software workaround for E-5 for more information on Self-ID related modifications.

2.5 E-5: Self-ID Packet Issues

Description of expected operation: After bus reset, the queues are flushed and a self-ID packet should be placed in the Read Request Queue. Refer to section 12.5.2.4 (Self-ID and PHY packets receive) in the "PDI1394L11 1394 AV Link Layer Controller" data sheet for the proper format of a self-ID packet.

Description of observed behavior: The following outcomes have been observed:

1. A correct self-ID packet.
2. A self-ID packet missing the 0x000000E0 header, but otherwise correct.
3. Nothing in the Read Request Queue (not even self-IDs).
4. Self-ID data delivered in the individual PHY packet format.
5. A 0x000000E0 header and 0x0000000D acknowledge code but no self-ID between them.

Solution or workaround: If there are no quadlets in the queue, i.e., RREQQQAV=0, then it can be assumed that there is a self-ID error, and a bus reset should be issued.

If there is a single quadlet and its value is 0x00000001, this would indicate that this is an isolated node missing the header 0x000000E0. If there is a single quadlet and its value is not 0x0000001 then a bus reset should be issued.

If there are two or more quadlets, then a check for a correct self-ID sequence should be conducted. If the first quadlet is 0x000000E0, discard it and continue. Now look for zero or more quadlet pairs where the second is the bitwise inverse of the first. When there is one quadlet left, and the pairing of each quadlet with a bitwise inverse has succeeded up to this point, then check the last quadlet for the value of 0x00000001. If this is the case, a correct self-ID sequence has been detected, otherwise do a bus reset.

Software Implementation: The above workaround can be implemented in the RDK by modifying function "ReadSelfID()" in module "Selfid.c" to include more checks. When an error is detected inside this function, a return value of FALSE is returned to the calling function. It is suggested that the calling function initiate a bus reset upon reception of return value FALSE.

SelfId.c - ReadSelfID()

```
BOOLEAN
ReadSelfID(BYTE xdata * size)
{
    BYTE i = 0;
    QUADLET quad;

     /* Go to the first self–ID quadlet in the queue */
    while ((AVLinkRead(ASYINTACK) & 0x0100) &&
        ((quad = AVLinkRead(RREQ)) == SELF_ID_START ||
        quad == OTHER_SELF_ID_END))
                        ;

    /* No self–ID available: one node network */
    if (!(AVLinkRead(ASYINTACK) & 0x0100))
    {

        *size = 0;
    }
    else
     /* Multiple nodes detected */
```

```
{
    /*
       Get all the self–ID packets
       and verify the inverse quadlet
       */
    /* Errata E-5 workaround
       do
       {
          if (quad != SELF_ID_END)
          {
             if (AVLinkRead(RREQ) != ~quad)
                    return FALSE;    ** inverse check failed **
             buffer[i++].quadlet = quad;
          quad = AVLinkRead(RREQ);
          }
       }
       while (i < MAX_1394_NODES && (AVLinkRead(ASYINTACK) & 0x0100) &&
       quad != SELF_ID_END);
       */

    /*
       Errata E-5 workaround.
       E-5: Self-ID Packet Issues.
       workarounds: see Errata E-5 workaround description
    */
       while (i < MAX_1394_NODES && (AVLinkRead(ASYINTACK) & 0x0100) &&
       quad != SELF_ID_END)
       {

       /* The next available quadlet must be the inverse quadlet */
        if (AVLinkRead(RREQ) != ~quad)
          return FALSE;  /* inverse check failed */
       buffer[i++].quadlet = quad;

       /* More self–ID quadlets or end quadlet? */
       if (AVLinkRead(ASYINTACK) & 0x0100)
          quad = AVLinkRead(RREQ);
       else
          return FALSE;       /* No more quadlets; No self-ID end quadlet */

       }

       *size = i;

    /* Look for an error in the rx request queue */
       if (RREQQRDERR())
       {
          CLR_RREQQRDERR();/* clear int. ack. bit */
          return FALSE;              /* request queue read error */
       }
       if (quad != SELF_ID_END)
          return FALSE;              /* buffer overflow or missing end */
  }
  return TRUE;
}
```

2.6 E-6: Incorrect Reception of PHY Packets

Description of expected operation: A proper PHY packet consists of two quadlets. These two quadlets are: (1) the synthesized header with value 0x000000E0, and (2) the first (non-inverted) quadlet as transmitted on the bus. Refer to section 12.5.2.4 (Self-ID and PHY packets receive) in the “PDI1394L11 1394 AV Link Layer Controller” data sheet for a description of a PHY packet.

Description of observed behavior: In some cases only the first header (value 0x000000E0) is transferred to the FIFO with the PHY packet being lost. The header will be prepended to and included as part of any subsequent received request packet causing that packet appear to be incorrectly framed. This has been seen to occur when a PHY packet arrives immediately (1) after a cycle-start packet, (2) after any packet which has been aborted due to insufficient FIFO space, or (3) after the retry protocol requires a busy_ack. The headers do not trigger interrupt RREQQQAV.

Solution or workaround: Disregard spurious header (value 0x000000E0) quadlets.

Software Implementation: The suggested implementation is restricted to Bus ID 0x3FF that is the current bus.

Following is the algorithm that could be used to detect and remove the above spurious header quadlets (value 0xE0).

When a request is received:

– Read the first quadlet.

– While the quadlet read is 0xE0, read the next quadlet; that is remove consecutive spurious headers

– At this point, the quadlet read is not 0xE0 and it is part of either an unformatted packet or a formatted packet. Examine the 2 most significant bits (msb).

Analysis: The unformatted packets have values of 00 for PHY configuration packet, 01 for a Link–On packet, or 10 for a self–ID packet. On the other hand, All the unformatted packets have the destination ID in the 16 msb, which consists of 10 bits for the Bus ID and 6 bits for the node ID. In case of a local bus, the Bus ID is 0x3FF thus the two most significant bits have a value of 11 for a formatted packet. Thus,

– If the 2 msbs have a value of 00 then treat the packet as a PHY configuration packet,

– If the 2 msbs have a value of 01 then treat the packet as a Link On packet,

– If the 2 msbs have a value of 10 then treat the packet as a self–ID packet,

– Else (that is the value is 11) the packet is a formatted packet, read the tCode and process the packet accordingly.

This algorithm can be applied to the RDK driver by modifying two functions in “Request.c”, “DecodePhyPacket()” and “RequestHandling()”, and modifying the prototype for “DecodePhyPacket()” in “Driver.h” as follows:

Driver.h

```
…
/*
   Errata E-6 workaround.
   E-6: Incorrect Reception of PHY Packets
   Workaround: see Errata Application note for E–6
   */
extern void DecodePhyPacket(QUADLET quad);
…
```

Request.c - DecodePhyPacket()

```
void
DecodePhyPacket(QUADLET quad)
{
    QDATA q;

    /* Get the Phy config quadlet and process it */

     /*
    Errata E-6 workaround.
    E-6: Incorrect Reception of PHY Packets
    Workaround: Discard spurious E0 packets (see Errata E–6):
    */
/* q.quadlet = ReqQueue(); */
q.quadlet = quad;
switch (q.byte[0] / 64)
{
        case 0:
            /* It is a Phy config packet */
            PutCharacter(PHYCONFIG_PACKET);
            PutQuad(q.quadlet);
            break;

        case 1:
            /* It is a link-on packet */
            PutCharacter(LINKON_PACKET);
            PutCharacter(q.byte[0] & 0x3F); /* phy ID */
            break;

        case 2:
            /* SelfID packet */
            PutCharacter(SELF_ID_PACKET);
            while (RREQQQAV())
                ReqQueue();
            break;

        default:
            /* unknown unformatted packet */
            PutCharacter(UNKNOWN_PHY_PACKET);
            while (RREQQQAV())
                ReqQueue();
    }
}
```

Request.c - RequestHandling()

```
void
RequestHandling(void)
{
    QDATA q;

    P1 |= 0x08; /* switch LED on */
    q.quadlet = ReqQueue();

 /*
    Errata E-6 workaround.
    E-6: Incorrect Reception of PHY Packets
    Workaround: Discard spurious E0 packets (see Errata E-6):
    */
    while ((q.byte[3] / 16) == T_PHY_PACKET)
    {
        q.quadlet = ReqQueue();
    }

    if ((q.byte[0] / 64) != 3)
    {
        /* Unformatted packet (Phy, Link on, or self ID packet):*/
        DecodePhyPacket(q.quadlet);
    }
    else
    {
            /* transaction code */
            switch ((q.byte[3] / 16))
            {

                /*
                case T_PHY_PACKET:
                    ** Unformatted packet **
                    DecodePhyPacket();
                    break;
                */

                case T_READ4_REQ:
                    /* Quadlet read request */
                    PutCharacter(QUADLET_READ_REQUEST);
                    QuadletReadReqHandling(q.quadlet);
                    break;

                case T_WRITE4_REQ:
                    /* Quadlet write request */
                    PutCharacter(QUADLET_WRITE_REQUEST);
                    QuadletWriteReqHandling(q.quadlet);
                    break;

                case T_LOCK_REQ:
                    /* Lock request */
                    PutCharacter(LOCK_REQUEST);
                    LockReqHandling(q.quadlet);
                    break;
```

```
            case T_READB_REQ:
                /* Block read request */
                PutCharacter(BLOCK_READ_REQUEST);
                BlockReadReqHandling(q.quadlet);
                break;

            case T_WRITEB_REQ:
                /* Block write request */
                PutCharacter(BLOCK_WRITE_REQUEST);
                BlockWriteReqHandling(q.quadlet);
                break;

            case T_ACK:
                /* Confirmation */
                PutCharacter(CONFIRMATION_PACKET);
                DecodeHeader(q.quadlet);
                break;

            default:
                PutCharacter(UNKNOWN_PACKET);     /* unknown t code */
        }

    } /* end else */
}
```

2.7 E-7: Incorrect Confirmation Packet Received when Broadcasting to an Even Numbered Bus ID

Description of expected operation: When a broadcast packet is sent to any bus ID other than the local bus (3FF), the resulting confirmation packet should contain an acknowledge code of ack_complete.

Description of observed behavior: When a broadcast packet is sent to an even numbered bus ID, then the resulting confirmation packet contains an incorrect acknowledge code of ack_missing.

Solution or workaround: Expect an acknowledge code of ack_missing for even numbered bus ID addresses.

Software Implementation: No software workaround.

2.8 E-8: Zero Length Lock Response Packets are Not Visible

Description of expected operation: All lock response packets should be visible at the receiver FIFO.

Description of observed behavior: Lock response packets with response code (rcode) value other than “resp_complete” (specified by the standard to be of zero length) are not visible on the receiver FIFO.

Solution or workaround: Typically the response code for a lock subaction is “resp_complete”. If a lock response is not received as expected, then it can be assumed that there was an error and the response code is other then “resp_complete”.

Software Implementation: No software workaround.

2.9 E-9: Wrong Detection of CIPTAGFLT and RCVBP Bits (Register IRXINTACK 0x4C)

Description of expected operation: When the Isochronous Receiver is disabled, interrupt bits CIPTAGFLT and RCVBP should be inactive.

Description of observed behavior: Occasionally, when the Isochronous Receiver is disabled, the CIPTAGFLT and RCVBP interrupt bits get set. When the Isochronous Receiver is enabled, then these bits work as expected.

Solution or workaround: Keep the receiver in RESET state when not in use; this will stop these bits from being set erroneously.

Software Implementation: Put the receiver in RESET mode at initialization, i.e., after bus or power reset, and when the receiver is stopped by the user. Any other time it would be the user's responsibility not to remove the receiver from the reset mode when it is not in use. This can be implemented in the RDK by modifying function "Init1394Com()" in "Main.c", and function "CommandHandling()" in "Command.c" as follows:

Main.c - Init1394Com()

```
void
Init1394Com(void)
{
   …

   /*
      Errata E-9 workaround.
      E-9: Wrong Detection of CIPTAGFLT and RCVBP Bits (Register IRXINTACK 0x4C).
      Workaround: Keep the isochronous receiver in reset mode when not in use.
   */
   SET_IRXRST();
}
```

Command.c - CommandHandling()

```
void
CommandHandling(char c)
{
      …
      case AVL_STOP_RX:  /* Stop reception */
         CLR_ENIRX();

         /*
         Errata E-9 workaround.
         E-9: Wrong Detection of CIPTAGFLT and RCVBP Bits (Register IRXINTACK 0x4C).
         Workaround: Keep the isochronous receiver in reset mode when not in use.
         */
         SET_IRXRST();
         break;
      …
}
```

2.10 E-10: Error Transmitting Large Asynchronous Packets

Description of expected operation: Packets of up to 64 quadlets (256 bytes) could be transmitted.

Description of observed behavior: When transmitting asynchronous requests (or responses), if the combined size of any two consecutive packets is greater than 64 quadlets, then interrupt bit RREQQFULL (or RRSPQFULL) gets set as soon as the 64th quadlet is written to the FIFO, and the transmission is halted.

Solution or workaround: When transmitting large packets make sure that the size of any two consecutive packets is less or equal to 64 quadlets. One way to do so is to transmit a small packet after transmitting a large packet. Another way is to keep the size of the transmitted packets 32 quadlets or less.

Software Implementation: Keep a global variable to store the size of the last sent asynchronous packet. Before another asynchronous packet is sent, check the size and add it to the global variable, if the total is greater than 64K then inform the user, otherwise send the packet and save its size in the global variable.

2.11 E-11: Wrong detection of TRSPQIDLE and TREQQIDLE when the FIFO is full

Description of expected operation: TRSPQIDLE and TREQQIDLE indicate that the transfer registers for the transmitter response and transmitter request queues, respectively, are idle. When a quadlet is written to either transfer register, the corresponding idle bit, TRSPQIDLE or TREQQIDLE is cleared. When the quadlet is transferred to the FIFO the idle bit gets set. This would be an indication to the application that the transfer register is ready to be written with the next quadlet.

Description of observed behavior: If a quadlet is written to the transfer register when the FIFO is full, the quadlet will remain in the transfer register and will not be transferred to the FIFO. The idle bits get set while they should remain cleared.

Solution or workaround: Before writing to the transfer register, the application should make sure that the FIFO is not full and that the previous write to the transfer register has been successfully transferred to the FIFO.

The application can check bits TREQQF and TRSPQF (bits 3 and 7 of ASYMEM 0x084) to determine if the FIFO is full. It can check bits REQQWR and TRSPQWR (bits 1 and 2 of ASYINTACK 0x0A0) to determine if the previous write to the transfer register has been successfully transferred to the FIFO.

Software Implementation: As listed above, the application should make sure that the FIFO is not full and that the last quadlet written to the transfer register has been successfully transferred to the FIFO. To determine if the FIFO is full, the application should either handle the FIFO full interrupts, TRSQFULL and TREQQFULL, and/or check bits TREQQF and TRSPQF right before writing a quadlet to the transfer register. To determine if the transfer register has been successfully transferred to the FIFO, the application could check bits TREQQWR and TRSPQWR right after a quadlet is written to that register. To implement this workaround in the RDK, the following functions have to be modified: SND_REQ_NEXT(), SND_REQ_LAST(), SND_RESP_NEXT(), and SND_RESP_LAST(). Everywhere these functions are called, the return values should be checked. Listed below are the modified functions and their prototypes, as well as the two utility functions CAN_WRITE_TO_REQQ() and CAN_WRITE_TO_RSPQ():

Driver.h -

```
…

/*
   Errata E-11 workaround.
   E-11: Wrong detection of TRSPQIDLE and TREQQIDLE when the
      FIFO is full
   Workaround: check bits TREQQF and TRSPQF to make sure that the
      FIFO is not full before writing to the transfer registers.
      Also check bits TREQQWR and TRSPQWR after a quadlet is
      written to the transfer register to make sure that it
      was successfully transferred to the queue.
      Functions SND_REQ_NEXT(), SND_REQ_LAST(),
      SND_RESP_NEXT(), and SND_RESP_LAST() have been modified
      and relocated to Packettx.c
   */
   /*
   #define SND_REQ_NEXT(quad)        AVLinkWrite(quad, TX_RQ_NEXT)
   #define SND_REQ_LAST(quad)        AVLinkWrite(quad, TX_RQ_LAST)
   #define SND_RESP_NEXT(quad)       AVLinkWrite(quad, TX_RP_NEXT)
   #define SND_RESP_LAST(quad)       AVLinkWrite(quad, TX_RP_LAST)
   */
   …

   /*
      Errata E-11 workaround.
      E-11: Wrong detection of TRSPQIDLE and TREQQIDLE when the
         FIFO is full
      Workaround: check bits TREQQF and TRSPQF to make sure that the
         FIFO is not full before writing to the transfer registers.
         Also check bits TREQQWR and TRSPQWR after a quadlet is
         written to the transfer register to make sure that it
         was successfully transferred to the queue.
         Functions SND_REQ_NEXT(), SND_REQ_LAST(),
         SND_RESP_NEXT(), and SND_RESP_LAST() have been modified
         and relocated to Packettx.c
   */
extern BOOLEAN SND_REQ_NEXT (QUADLET quad);
extern BOOLEAN SND_REQ_LAST (QUADLET quad);
extern BOOLEAN SND_RESP_NEXT(QUADLET quad);
extern BOOLEAN SND_RESP_LAST(QUADLET quad);
```

Packettx.c -
CAN_WRITE_TO_REQQ(), CAN_WRITE_TO_RSPQ(),
SND_REQ_NEXT(), SND_REQ_LAST(),
SND_RESP_NEXT(), and SND_RESP_LAST()

```
/*****************************************************************
* Functions:  CAN_WRITE_TO_REQQ(), CAN_WRITE_TO_RSPQ(),
*           SND_REQ_NEXT(), SND_REQ_LAST(),
*           SND_RESP_NEXT(), and SND_RESP_LAST()
*
* Description:    Send a quadlet to the transfer registers.
*
* Arguments:  quad:   The quadlet to be written to the transfer register
*
* Return:         TRUE if the write was successful, FALSE otherwise.
*
*****************************************************************/
/*
   Errata E-11 workaround.
   E-11: Wrong detection of TRSPQIDLE and TREQQIDLE when the
      FIFO is full
   Workaround: Check bits TREQQF and TRSPQF to make sure that the FIFO
      is not full before writing to it. Also check bits TREQQWR
      and TRSPQWR after a quadlet is written to the transfer register
      to make sure that it was successfully transferred to the queue.
      The functions below were relocated out of Driver.h and into
      this module.
   */

BOOLEAN
CAN_WRITE_TO_REQQ(void)
{

   if ( LinkRead(ASYINTACK) & 0x00000010 )
      return FALSE;

   return TRUE;
}

BOOLEAN
CAN_WRITE_TO_RSPQ(void)
{

   if ( LinkRead(ASYINTACK) & 0x00000020 )
      return FALSE;

   return TRUE;
}
```

```
BOOLEAN
SND_REQ_NEXT(QUADLET quad)
{

   if (CAN_WRITE_TO_REQQ())
   {
      AVLinkWrite(quad, TX_RQ_NEXT);
      if (LinkRead(ASYINTACK) & 0x00000001 )
         return TRUE;
   }

   return FALSE;
}

BOOLEAN
SND_REQ_LAST(QUADLET quad)
{

   if (CAN_WRITE_TO_REQQ())
   {
      AVLinkWrite(quad, TX_RQ_LAST);
      if (LinkRead(ASYINTACK) & 0x00000001 )
         return TRUE;
   }

   return FALSE;
}

BOOLEAN
SND_RESP_NEXT(QUADLET quad)
{

   if (CAN_WRITE_TO_RSPQ())
   {
      AVLinkWrite(quad, TX_RP_NEXT);
      if ( LinkRead(ASYINTACK) & 0x00000002 )
         return TRUE;
   }

   return FALSE;
}

BOOLEAN
SND_RESP_LAST(QUADLET quad)
{

   if (CAN_WRITE_TO_RSPQ())
   {
      AVLinkWrite(quad, TX_RP_LAST);
      if ( LinkRead(ASYINTACK) & 0x00000002 )
         return TRUE;
   }

   return FALSE;
}
```

2.12 E-12: Dual Phase Fairness Interval not compliant with 1394-1995 standards

Description of expected operation: Section 3.6.2.4 of the 1394-1995 standard specifies that dual phase destination nodes shall assume all retry_A (or retry_B) subactions have been sent when four fairness intervals pass with no retry_A (or retry_B) subactions having been busied.

Description of observed behavior: For inbound transactions, two fairness intervals are implemented instead of four.

Solution or workaround: The application can specify that only single phase protocol should be used for inbound. This can be done by setting the Busy Control field (BSYCTRL: bits 27–29 of LNKCTL 0X004) to 011.

Software Implementation: Set the value of BSYCTRL at initialization of the link layer; that is after bus and power resets. This can be accomplished by modifying “Driver.h” and function “InitLink()” in “Main.c” as follows:

Driver.h

```
…
   /*
      Errata E-12 workaround.
      E-12:  Dual Phase Fairness Interval not compliant with 1394–1995 standards.
      Workaround: Set the inbound to single phase after bus and power resets.
   */
#define SetSinglePhaseOnInbound()   AVLinkWrite(AVLinkRead(LNKCTL) | 0x18000000, LNKCTL)
…
```

Main.c - InitLink()

```
void
InitLink(void)
{
   …
    /*
      Errata E-12 workaround.
      E-12:  Dual Phase Fairness Interval not compliant with 1394-1995 standards.
      Workaround: Set the inbound to single phase after bus and power resets.
   */
   SetSinglePhaseOnInbound();
}
```

Software workarounds for Philips PDI1394L11 Errata

AN2451

Definitions

Short-form specification — The data in a short-form specification is extracted from a full data sheet with the same type number and title. For detailed information see the relevant data sheet or data handbook.

Limiting values definition — Limiting values given are in accordance with the Absolute Maximum Rating System (IEC 134). Stress above one or more of the limiting values may cause permanent damage to the device. These are stress ratings only and operation of the device at these or at any other conditions above those given in the Characteristics sections of the specification is not implied. Exposure to limiting values for extended periods may affect device reliability.

Application information — Applications that are described herein for any of these products are for illustrative purposes only. Philips Semiconductors make no representation or warranty that such applications will be suitable for the specified use without further testing or modification.

Disclaimers

Life support — These products are not designed for use in life support appliances, devices or systems where malfunction of these products can reasonably be expected to result in personal injury. Philips Semiconductors customers using or selling these products for use in such applications do so at their own risk and agree to fully indemnify Philips Semiconductors for any damages resulting from such application.

Right to make changes — Philips Semiconductors reserves the right to make changes, without notice, in the products, including circuits, standard cells, and/or software, described or contained herein in order to improve design and/or performance. Philips Semiconductors assumes no responsibility or liability for the use of any of these products, conveys no license or title under any patent, copyright, or mask work right to these products, and makes no representations or warranties that these products are free from patent, copyright, or mask work right infringement, unless otherwise specified.

Philips Semiconductors
811 East Arques Avenue
P.O. Box 3409
Sunnyvale, California 94088–3409
Telephone 800-234-7381

Date of release: 08-98

Document order number: 9397 750 04232

Let's make things better.

Philips Semiconductors

PHILIPS