CP3UB17/CP3CN17

User Information Sheet

September 21, 2003

National Semiconductor
The Sight & Sound of Information

This user information document discloses known problems and limitations with the CP3UB17/CP3CN17. This document also provides information on potential user issues which have not yet been incorporated into product documentation and planned design enhancements for future revisions of the devices.

Number: 1

Name: ACCESS.bus Interface Module Bus Error

Description: A Bus Error (BER) may occur during a write transaction if the data register is written at a very specific time. The module generates one system-clock cycle setup time of SDA to SCL vs. the minimum time of the clock divider ratio.

Workaround: The problem can be masked within the driver by dynamically dividing-by-half the SCL width immediately after the slave address is successfully sent and before writing to the ACBSDA register. This has the effect of forcing SCL into the stretch state.

The following code example is the relevant segment of the ACCESS.bus driver addressing this issue.

```
/*%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
; NAME: ACBRead     Reads "Count" byte(s) from selected I2C Slave.  If read address differs from previous
;                   Read or Write operation (as recorded in NextAddress), a "dummy" write transaction is
;                   initiated to reset the address to the desired location.  This is followed by a repeated
;                   Start sequence and the Read transaction.  All transactions begin with a call to ACBStartX
;                   which sends the Start condition and Slave address.  Checks for errors throughout process.
;
; PARAMETERS:   UBYTE   Slave   -   Slave Device Address. Must be of format 0xXXXX0000
;               UWORD   Addrs   -   Byte/Array address (extended addressing mode uses two byte address)
;               UWORD   Count   -   Number of bytes to read
;               UBYTE   *buf    -   Pointer to receive buffer
;
; CALLS:    ACBStartX
;
; RETURNED: error status
;%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%*/

UWORD   ACBRead (UBYTE Slave, UWORD Addrs, UWORD Count, UBYTE *buf)
{
    ACB_T   *acb;
    UBYTE   err, *rcv;
    UWORD   Timeout;

    acb =   (ACB_T*)ACB_ADDRESS;                    /* Set pointer to ACB module                            */
                                                    /* If the indicated address differs from the last       */
    if (Addrs != NextAddress) {
                                                    /* recorded access (i.e. Random Read), we must first    */
                                                    /* send a "dummy" write to the desired new address..    */
        NextAddress =   Addrs;                      /* Update last address placeholder                      */


    KeyInit();
                    KBD_OUT &= ~BIT0;

                                                    /* Send start bit and Slave address...                  */
        if ((err = ACBStartX (Slave | (Addrs >> 7 & 0x0E), ACB_WRITE, 0)))

            return (err);                           /* If unsuccessful, return error code                   */

 //   KBD_OUT &= ~BIT0;

        acb->ACBsda =   (UBYTE)Addrs;               /* Send new address byte                                */

KBD_OUT &= ~BIT0;

        Timeout =   1000;                           /* Set timeout                                          */
                                                    /* Wait for xmitter to be ready...zzzzzzzzz             */
        while (!(acb->ACBst & ACBSDAST) && !(acb->ACBst & ACBBER) && Timeout--);

                                        if (acb->ACBst & ACBBER) {
/* If a bus error occurs while sending address, clear   */

            acb->ACBst  |=  ACBBER;                 /* the error flag and return error status               */

            return (ACBERR_COLLISION);
        }

 KBD_OUT &= ~BIT0;

        if (!Timeout)                               /* If we timeout, return error                          */
            return (ACBERR_TIMEOUT);
    }
                                                    /* (Re)Send start bit and Slave address...              */
    if ((err = ACBStartX (Slave | (Addrs >> 7 & 0x0E), ACB_READ, Count)))
                                                    /* If error, return                                     */
        return (err);

    rcv =   buf;                                    /* Get address of read buffer                           */
                                                    /* Read Count bytes into user's buffer                  */
    while (Count) {

        if (Count-- == 1)                           /* If this the final byte, or only one requested, send  */
            acb->ACBctl1    |=  ACBACK;             /* the NACK bit after reception                         */

        Timeout =   1000;                           /* Set timeout                                          */

        while (!(acb->ACBst & ACBSDAST) && Timeout--);

        if (!Timeout)                               /* Timed out??                                          */
                                                    /* YES - return error                                   */
            return (ACBERR_TIMEOUT);

        *rcv++  =   acb->ACBsda;                    /* NO - Read byte from Recv register                    */
                                                   /* Adjust current address placeholder                   */
        NextAddress++;
```

```
    }

    acb->ACBctl1    |=  ACBSTOP;                    /* Send STOP bit                                        */
                                                    /* Return success status....                            */
    return (ACB_NOERR);
}

/*%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
; NAME: ACBStartX   Initiates an ACB bus transaction by sending the Start bit, followed by the Slave address
;                   and R/W flag.  Checks for any ACB errors throughout this sequence and returns status.
;
; PARAMETERS:   UBYTE   Slave   -   I2C address of Slave device
;               UBYTE   R_nW    -   Read/Write flag (0x01 or 0x00)
;               UWORD   Count   -   Desired number of bytes (read/write)
;
; CALLS:
;
; RETURNED: error/success
;%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%*/

UWORD   ACBStartX (UBYTE Slave, UBYTE R_nW, UWORD Count)
{
    ACB_T   *acb;
    UWORD   Timeout;
                                                    /* Get address of ACB module                            */
    acb =   (ACB_T*)ACB_ADDRESS;
                                                    /* If Bus is Busy and we're NOT the Master, return err  */
    if (acb->ACBcst & ACBBB && !(acb->ACBst & ACBMASTER))
        return (ACBERR_NOTMASTER);
                                                    /* If we're good to go, send Start condition            */
    acb->ACBctl1    |=  ACBSTART;
                                                    /* Check if we're the Bus Master with timeout           */
    Timeout =   100;

    while (!(acb->ACBst & ACBSDAST) && Timeout--)   /* Related to bus error problem                         */
    {
        if (acb->ACBst & ACBBER) {                  /* If collision occurs, clear error and return status   */

            acb->ACBst  |=  ACBBER;

            return (ACBERR_COLLISION);
        }
    }

    if (!Timeout)                                   /* If timeout, we must NOT be the Master...signal error */
        return (ACBERR_NOTMASTER);
                                                    /* Now, send the address and R/W flag...                */
    acb->ACBsda =   Slave | R_nW;                   /* Send address and R/W flag                            */

    Timeout =   1000;                               /* Failsafe for lockup                                  */
                                                    /* Wait for address to be sent and ACK'd                */
    while (!(acb->ACBst & ACBSDAST) &&
                                            !(acb->ACBst & ACBNEGACK)&&
                                            --Timeout) {

        if (acb->ACBst & ACBBER) {                  /* If a bus error occurs while sending address, clear   */

            acb->ACBst  |=  ACBBER;                 /* the error flag and return error status               */

            return (ACBERR_COLLISION);
        }
    }

KBD_OUT |= BIT0; // OScope marker

    if (!Timeout)                                   /* If timeout, signal error                             */
        return (ACBERR_TIMEOUT);
                                                    /* Or if Slave does not reply, report busy/error        */
    else if (acb->ACBst & ACBNEGACK)
        return (ACBERR_NEGACK);
                                                    /* Otherwise return success                             */
    else {

        return (ACB_NOERR);
    }
```

Number: **2**

Name: **Advanced Audio Interface Asynchronous Mode**

Description: The AAI asynchronous mode only works if it is receiving the frame syncs and not generating them.

Workaround: In asynchronous mode, configure the AAI to receive frame syncs.

Number: **3**

Name: **Advanced Audio Interface**

Description: While the AAI is active, it can lock up if the receive FIFO is cleared with the Clear Receive FIFO (CRF) bit, the transmit FIFO is cleared with the Clear Transmit FIFO (CTF) bit, or the AAI is disabled by clearing the AGCR.AAIEN bit.

Workaround: While the AAI is active, never clear the receive FIFO with the CRF bit or clear the transmit FIFO with the CTF bit. To disable the AAI, clear the ARSCR.RXSA and ATSCR.TXSA bits, then wait 10 receive/transmit clock cycles before clearing the AGCR.AAIEN bit.

Number: **4**

Name: **CVSD/PCM FIFO**

Description: If the CVSDIN register (Rx FIFO) is read while containing no data, an irreversible lockup in the FIFO logic can occur, rendering it unusable until reset.

Workaround: The interrupt routine MUST use the value reported in the CVOUTST field of the CVSDIN register to determine the actual number of words in the Rx FIFO. Under no circumstances should one assume that the CVNF flag always indicates that five words are available to be read.

Number: **5**

Name: **USB 2.0 Compliance**

Description: This entry is to serve as a reminder that the USB as implemented is certified for USB 1.1 compliance.

Workaround: An application note is available which describes an external circuit to achieve 2.0 compliance.

Number: **6**

Name: **UART RX Lockup**

Description: USART can lock up in the presence of noise or an incorrect RX rate.

At a high level, what is happening is that the START DETECT state machine detects a START bit but a signal which is meant to enable this to the USART RX state machine happens in a different clock cycle from where it is required. So, the START DETECT state machine sits and waits for a STOP from the USART RX state machine, but the USART RX state machine is sitting in RX_IDLE mode because it has not seen the detection of the START bit.

Workaround: Fixed in the next silicon revision.