CP/SP Communication and Organization

This document discusses the developments and discoveries regarding serial communication between CP and SP boards following the latest revision.

Flagstaff, AZ
April 2nd, 2014

Edited by

Michael Middleton
Northern Arizona University
Wireless Networks Research Laboratory

Contents

1 Overview

An investigation of the WiSARD UART communication implementation has yielded performance information regarding the accuracy and precision of data sampling and has provided new insight into the limitations of the maximum baud rate possible on current WiSARD hardware. The previous revision of the UART communication implementation between CP and SP contained latencies in the transmission and sampling of serial messages due to an approach that was functional but was not optimal for fast and efficient communication.

New revisions to the communication code involving the restructuring of the Port2 and Timer0 A0 interrupt service routines and the behavior of functions within the protocol have allowed for a communication system resilient to errors and mis-sampling, as well as an overall improved form which utilizes good software engineering practices and robust design. Figure 1 shows the sampling of a single byte from a command packet sent from CP to an SP light board. The activity represented by the light blue is the CP transmit line and the yellow activity is a trace that depicts the time at which the SP samples the line. The rising and falling edges of the trace indicate the precise moment that the code enters the switch-case statement to processes the current bit, located in the receiveByte function (Appendix B.1, line 339).

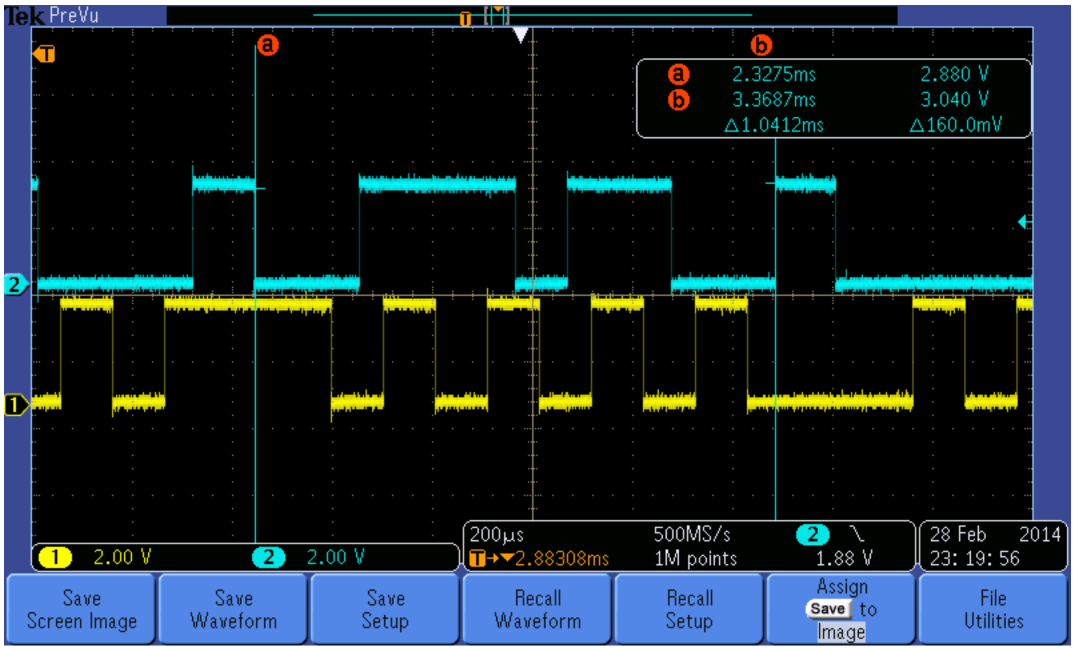

Figure 1: CP to SP byte transmission

Figure 2 depicts the transmission and sampling of an entire 32-bit message from the same command packet as Figure 1. These images show an overall improvement in sampling accuracy and precision over the previous revision of the communication code. On average, samples are taken within 1us of the middle of every bit across all baud rates, whereas the code from the previous revision would often be within 2us of the center of a bit.

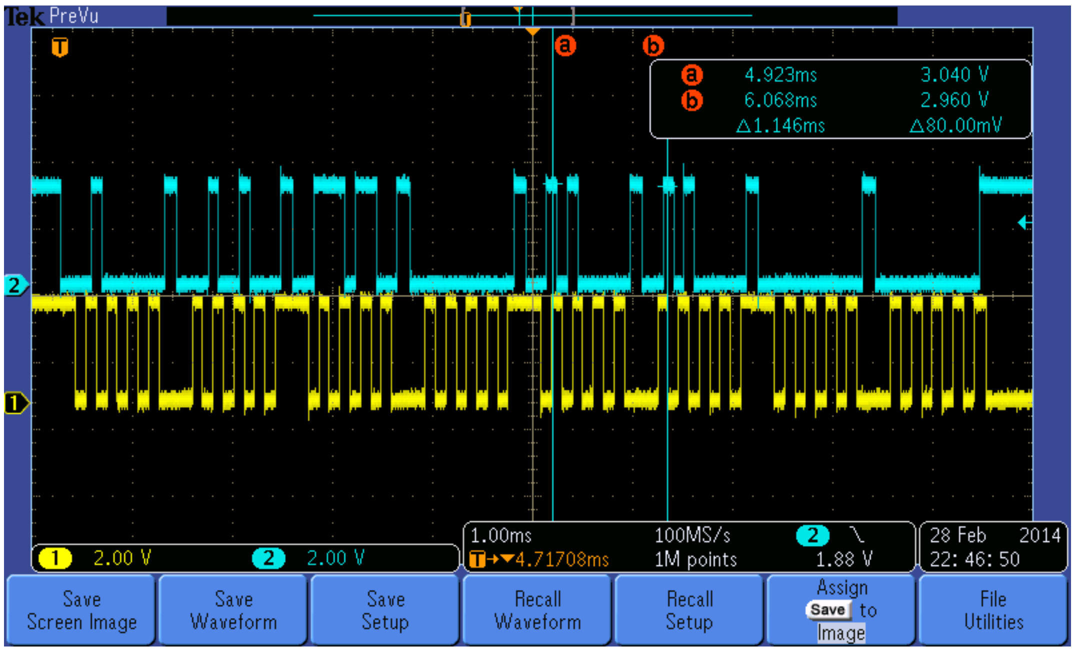

Figure 2: CP to SP 32-bit packet transmission

2 Code Reorganization

The previous revision of the UART communication implementation between CP and SP contained latencies from a few different sources, most significantly from the Timer0 A0 interrupt service routine. Other latencies and behavioral issues in the communication code developed from the manner in which data messages were received within the Timer0 A0 ISR, the mishandling of the wake up times, and the lack of a consistent generalized approach to sampling in the center of each bit.

2.1 Interrupt Service Routines

Timer0 A0 uses a 4MHz SMCLK to count up to the length of a single bit, defined by the current baud rate, and then triggers a Timer0 A0 interrupt. In the transmission code, when the Timer0 A0 interrupt is triggered, the TX line will be pulled high or low depending upon the value of the next bit in the message to be sent. The receiver portion uses the same interrupt to sample the TX line, determine the location of the bit in the message, and shift the bit to its correct location in the received message on the RX buffer. Previously, the code within the ISR contained the logic used to determine whether the hardware was sending or receiving a message, and then handled the message bit appropriately (Appendix A.3). Even though the bits were being handled as soon as the interrupt was triggered, issues arose from executing this logic within the ISR. At the time the interrupt is triggered, the timer is halted and execution moves to the code within in the ISR. The 2.5us time associated with waking the MCU from LPM0 and sequentially evaluating the message logic effectively stalls the start of Timer0 A0 counting to the next bit until execution returns from the ISR. This approach to sending and receiving messages is problematic because latency from executing the logic within the ISR compounds with each bit in a byte until the TX and RX re-synchronize at the falling edge of a start bit.

To remedy the negative effects of structuring the code in this manner, a new approach was developed which reorganized the way in which the code was executed. The first step was to remove the message logic from the Timer0 A0 ISRs and place them into sendByte() and receiveByte() functions. These functions would be called once for each byte transmission or reception and rest in LPM0 while waiting for the next bit to be sent or received. The sole operation of the Timer0 A0 ISR, now that the message logic has been removed, is to simply wake the MCU from LPM0 and return to program execution (Appendix A.4). The ISR no longer needs to check the state of the hardware either, since the waking from LPM0 is required whether the hardware is in transmission or reception. This change exhibits good form within interrupt driven design as the ISRs are as simple as possible, and make the smallest impact on the continuity between hardware components. Additionally, these changes make the code more modular in its design. Byte transmission and reception, as well as waking from LPM0 are now decoupled and can be altered independent of the other components. This will help in making future edits and revisions simpler to debug and easier to navigate.

2.2 Generalized Delay Solution

Further changes made to the communication code involved mitigating delays associated with a receiver sampling the first data bit of a byte. Following the falling edge of the TX-line which indicates the start of a byte transmission, the receiver would ideally wait the duration of the start bit, and half the duration of a second bit so that the first sample will be taken in the middle of the data bits, leaving the largest possible error tolerance. In the previous revision, each baud rate used hard-coded values for the number of clock cycles on the 4MHz SMCLK which would count to approximately one half of the bit length for the current baud rate (Appendix B.2). There was no uniformity to the way in which these values were achieved and they did not completely account for the full wake up time from LPM3 in the first byte of a new message. The time required to wake from LPM3 as listed in the MSP430x5xx family users guide is approximately 9us.

To handle this problem with a more generalized solution, a single formula was used to calculate the delay values and account for the 9us wake up time for LPM3. By implementing the formula [(1.5 * BaudRateControl) - 36] clock cycles calculated for a 4MHz SMCLK, a generalized delay for the sampling as well as compensating for MCU wake up time was achieved across the baud rates 115200, 57600, 19200, and 9600 (Appendix B.3). This formula is generalized such that it can be used to calculate the half bit delay for any baud rate with Timer0 A0 sourced from a 4MHz SMCLK. This formula works because the 36 clock cycles at 4MHz accounts for the same 9us delay present regardless of the current baud rate.

3 Faster Baud Rates

Current WiSARD hardware supports the possibility of baud rates faster than 115200, though doing so would require altering the approach to sending and receiving messages. The current approach places the MCU in LPM0 while Timer0 A0 counts on the appropriate interval. Waking the MCU from LPM0 takes 2.5us, regardless of baud rate. The baud rate 230400 is such that the length of a single bit is 4.34us. The LPM0 wakeup time is greater than the half bit delay required for 230400 at the start of each byte. Accommodating the half bit delay would require that for a baud rate of 230400, the MCU would need to remain on for the duration of the message on both the transmission and reception. Additionally, the 9us delay necessary to wake the MCU from LPM3 would require additional wait time at the front of a message. The length of the start bit is adequate to accommodate the wakeup time for all baud rates up to 115200 but 230400 would need at least 9us, which would require slightly longer than 2 start bits. Adding additional bits to the front of a packet would mean that the communication is no longer within the formal specification of the UART protocol, meaning that interfacing with third party hardware or software which communicates via UART would no longer be a possibility. Since 115200 is the fastest baud rate incorporating low power modes while maintaining UART communication specifications, the decision was made to forsake the implementation of faster baud rates for the time being.

A CP Code Modifications

A.1 SP.c (Current Revision)

This function contains the majority of the logic for byte transmission that was removed from the Timer0 A0 interrupt service routine.

SendByte Function

title

```
///////////////////////////////////////////////////////////////////////////////
//! \brief Sends a byte via the software UART
//!
//! This function pushes \e ucChar into the global TX buffer, where the
//! TimerA ISR can access it. The system drops into LPM0, keeping the SMCLK
//! alive for TimerA. The ISR handles the start and stop bits as well as the
//! baud rate. This is a blocking call and will not return until the software
//! has sent the entire message.
//!   \param ucChar The 8-bit value to send
//!   \return None
///////////////////////////////////////////////////////////////////////////////
static void vSP_SendByte(uint8 ucTXChar)
{

    uint8 ucParityBit;
    uint8 ucBitIdx;
    uint8 ucTXBitsLeft;

    // If we are already busy, return so as not to screw it up
    if (g_ucCOMM_Flags & COMM_TX_BUSY)
        return;

    P_RX_IE &= ~S_ActiveSP.m_ucActiveSP_BitRx;

    // Indicate in the status register that we are now busy
    g_ucCOMM_Flags |= COMM_TX_BUSY;

    // Calculate the parity bit prior to transmission
    ucParityBit = 0;
    for(ucBitIdx = 0; ucBitIdx < 8; ucBitIdx++)
    {
        ucParityBit ^= ((ucTXChar & 0x01)>>ucBitIdx);
    }

    // Reset the bit count so the ISR knows how many bits left to send
    ucTXBitsLeft = 0x0A;

    TA0CCR0 = g_unCOMM_BaudRateControl;
    // Starts the counter in 'Up-Mode'
    TA0CTL |= TACLR | MC_1;

    while (g_ucCOMM_Flags & COMM_TX_BUSY)
    {
        switch (ucTXBitsLeft)
        {
            case 0x00:
                // Last bit is stop bit, return to idle state
                *S_ActiveSP.m_ucActiveSP_RegOut |= S_ActiveSP.m_ucActiveSP_BitTx;
                __bis_SR_register(GIE + LPM0_bits);
                g_ucCOMM_Flags &= ~COMM_TX_BUSY;
            break;

            case 0x01:
                if (ucParityBit)
                    *S_ActiveSP.m_ucActiveSP_RegOut |= S_ActiveSP.m_ucActiveSP_BitTx;
                else
                    *S_ActiveSP.m_ucActiveSP_RegOut &= ~S_ActiveSP.m_ucActiveSP_BitTx;

                __bis_SR_register(GIE + LPM0_bits);
            break;

            case 0x0A:
                if (ucTXChar & 0x01)
                    *S_ActiveSP.m_ucActiveSP_RegOut &= ~S_ActiveSP.m_ucActiveSP_BitTx;
                else
                    *S_ActiveSP.m_ucActiveSP_RegOut &= ~S_ActiveSP.m_ucActiveSP_BitTx;

                // First bit is start bit
//              *S_ActiveSP.m_ucActiveSP_RegOut &= ~S_ActiveSP.m_ucActiveSP_BitTx;
                __bis_SR_register(GIE + LPM0_bits);
            break;

            default:
                // For data bits, mask to get correct value and the shift for next time
                if (ucTXChar & 0x01)
                    *S_ActiveSP.m_ucActiveSP_RegOut |= S_ActiveSP.m_ucActiveSP_BitTx;
                else
                    *S_ActiveSP.m_ucActiveSP_RegOut &= ~S_ActiveSP.m_ucActiveSP_BitTx;
                ucTXChar >>= 1;
                __bis_SR_register(GIE + LPM0_bits);
            break;
        }

        // Decrement the total bit count
        ucTXBitsLeft--;

    }

    P_RX_IE |= S_ActiveSP.m_ucActiveSP_BitRx;

    // Stop the timer
    TA0CTL &= ~(MC0 | MC1 | TAIFG);

    P3OUT |=  BIT4;
}

////////////////////////////////////////////////////////////////////////////////
```

This function contains the majority of the logic for byte reception that was removed from the Timer0 A0 interrupt service routine.

ReceiveByte Function

title

```
////////////////////////////////////////////////////////////////////////////////
//! \brief Receives a byte via the software UART
//!
//!   \param None
//!   \return None
////////////////////////////////////////////////////////////////////////////////
static uint8 ucSP_ReceiveByte(void)
{
    uint8 ucRXBitsLeft;
    uint8 ucParityBit;  // The calculated parity bit
    uint8 ucRxParityBit; // The received parity bit
    uint8 ucBitIdx;


    // Set up timer
    TA0CTL = (TASSEL_2 | TACLR);
    TA0CCTL0 &= ~CCIE;
    TA0CCR0 = 0xFFFF;

    // Wait for the port interrupt to exit LPM and continue
    TA0CCTL0 |= CCIE;
    TA0CTL |= MC_1;

    // shut off MCU
    LPM0;

    P2IE &= ~0x0F; // Disable interrupts on the RX pin
    TA0CTL = (TASSEL_2 | TACLR);
    TA0CCTL0 &= ~CCIE;

    //If a port interrupt was received then we are in the RX active state else exit
    if( !(g_ucCOMM_Flags & COMM_RX_BUSY))
    {
        return COMM_ERROR;
    }

    ucRXBitsLeft = 0x09;

    // Start timer and delay for half a bit
    TA0CCR0 = g_unCOMM_BaudRateDelayControl;
    TA0CTL |= MC_1;
    while (!(TA0CTL & TAIFG));
    TA0CTL &= ~TAIFG;

    // Set up timer for comm. at the baud rate
    TA0CTL = (TASSEL_2 | TACLR);
    TA0CCTL0 = CCIE;
    TA0CCR0 = g_unCOMM_BaudRateControl;

    // Start the timer
    TA0CTL |= MC_1;

    while(g_ucCOMM_Flags & COMM_RX_BUSY)
    {
            switch (ucRXBitsLeft)
            {
                case 0x00:
                    // There are no bits left , so lets reset all the values and stop timer
                    TA0CTL = (TASSEL_2 | TACLR);
                    P_RX_IFG &= ~S_ActiveSP.m_ucActiveSP_BitRx;
                    P_RX_IE |= S_ActiveSP.m_ucActiveSP_BitRx;
                    g_ucCOMM_Flags &= ~COMM_RX_BUSY;
                    // Increment index for next byte
                    g_ucRXBufferIndex++;
                break;

                // Parity Bit
                case 0x01:
                    if (P_RX_IN & S_ActiveSP.m_ucActiveSP_BitRx)
                        ucRxParityBit = 1;
                    else
                        ucRxParityBit = 0;

                    // shut off MCU
                    LPM0;
                break;

                // Last data bit no shift
                case 0x02:
                    if (P_RX_IN & S_ActiveSP.m_ucActiveSP_BitRx)
                        g_ucaRXBuffer[g_ucRXBufferIndex] |= 0x80;
                    else
                        g_ucaRXBuffer[g_ucRXBufferIndex] &= ~0x80;

                    // shut off MCU
                    LPM0;
                break;

                default:
                    if (P_RX_IN & S_ActiveSP.m_ucActiveSP_BitRx)
                        g_ucaRXBuffer[g_ucRXBufferIndex] |= 0x80;
                    else
                        g_ucaRXBuffer[g_ucRXBufferIndex] &= ~0x80;

                    g_ucaRXBuffer[g_ucRXBufferIndex] >>= 1;
                    LPM0;
                break;

            }
            ucRXBitsLeft--;

    }

    // Check Parity
//  ucParityBit = 0;
//  for(ucBitIdx = 0; ucBitIdx < 8; ucBitIdx++)
//  {
//      ucParityBit ^= ((g_ucaRXBuffer[g_ucRXBufferIndex] & 0x01)>>ucBitIdx);
//  }

    //Todo move the parity check to the read from buffer function to keep comm simple
//  if(ucParityBit != ucRxParityBit)
//      return COMM_PARITY_ERR;

    return COMM_OK;
}

/////////////////////////////////////////////////////////////////////////////
//! \brief Shuts off the software modules
//!
//! This shuts down TimerA and disables all of the interrupts used. The vSP_ShutdownComm()
//! function must be used when switching between SP boards otherwise the communication
//! will fail.
```

This function contains the logic necessary for the receiver to adapt its anticipated message size based upon the packet header which it receives.

WaitForMessage Function

title

```
uint8 ucSP_WaitForMessage(void)
{

  // Set the size of the received message to the minimum
    g_ucRXMessageSize = SP_HEADERSIZE;

    // Wait to receive the message
    do // saves 3 cycles as opposed to while
    {

        if(ucSP_ReceiveByte())
        {
            return COMM_ERROR;
        }

        // If we have received the header of the message, update the RX message to
        // the size of the message received
        if (g_ucRXBufferIndex == SP_HEADERSIZE)
        {
            g_ucRXMessageSize = g_ucaRXBuffer[0x01];

            // Range check the g_ucRXMessageSize variable
            if (g_ucRXMessageSize > MAX_SP_MSGSIZE || g_ucRXMessageSize < SP_HEADERSIZE)
                return 0x04;
        }


    }while (g_ucRXBufferIndex != g_ucRXMessageSize);

    // No message received
    if (g_ucRXBufferIndex == 0)
    {
        g_ucRXMessageSize = 0;
        return COMM_ERROR;
    }

//success
    return 0;
}
```

A.2 SP.h (Current Revision)

The following code demonstrates the current definitions for the baud rate delay values which utilizes the generalized delay cycle calculation formula.

title

```
//! \def BAUD_115200_DELAY
//! \brief Timer count for specific data rate delay, computed for 4Mhz SMCLK.
#define BAUD_115200_DELAY  BAUD_115200 + BAUD_115200/2 - 37//0x0010//Slightly low to account
     for the set-up cycles that we delayed. At these times that's necessary...
//! \def BAUD_57600_DELAY
//! \brief Timer count for specific data rate delay, computed for 4Mhz SMCLK.
#define BAUD_57600_DELAY   BAUD_57600 + BAUD_57600/2 - 36
//! \def BAUD_19200_DELAY
//! \brief Timer count for specific data rate delay, computed for 4Mhz SMCLK.
#define BAUD_19200_DELAY   BAUD_19200 + BAUD_19200/2 - 36
//! \def BAUD_9600_DELAY
//! \brief Timer count for specific data rate delay, computed for 4Mhz SMCLK.
#define BAUD_9600_DELAY    BAUD_9600 + BAUD_9600/2 - 36
```

A.3 irupt.c (Previous Revision)

The following code demonstrates the previous behavior of the Timer0 A0 interrupt service routine.

title

```
#pragma vector=TIMER0_A0_VECTOR
```

```
__interrupt void TIMER0_A0_ISR(void)
{

    if (g_ucCOMM_Flags & COMM_TX_BUSY)
    {
        switch (g_ucTXBitsLeft)
        {
            case 0x00:
                // If there are no bits left, then return to function call
                __bic_SR_register_on_exit(LPM0_bits);
            break;
            case 0x01:
                // Last bit is stop bit, return to idle state
                *S_ActiveSP.m_ucActiveSP_RegOut |= S_ActiveSP.m_ucActiveSP_BitTx;
                __bic_SR_register_on_exit(LPM0_bits);
            break;
            case 0x0A:
                // First bit is start bit
                *S_ActiveSP.m_ucActiveSP_RegOut &= ~S_ActiveSP.m_ucActiveSP_BitTx;
            break;
            default:
                // For data bits, mask to get correct value and the shift for next time
                if (g_ucTXBuffer & 0x01)
                    *S_ActiveSP.m_ucActiveSP_RegOut |= S_ActiveSP.m_ucActiveSP_BitTx;
                else
                    *S_ActiveSP.m_ucActiveSP_RegOut &= ~S_ActiveSP.m_ucActiveSP_BitTx;
                g_ucTXBuffer >>= 1;
            break;
        }
        // Decrement the total bit count
        g_ucTXBitsLeft--;
    }
    if (g_ucCOMM_Flags & COMM_RX_BUSY)
    {
        switch (g_ucRXBitsLeft)
        {
            case 0x00:
                // There are no bits left, so lets reset all the values and stop timer
                TA0CTL &= ~(MC0 | MC1);
                P_RX_IE |= S_ActiveSP.m_ucActiveSP_BitRx;
                P_RX_IFG &= ~S_ActiveSP.m_ucActiveSP_BitRx;
                g_ucRXBufferIndex++;
                g_ucCOMM_Flags &= ~COMM_RX_BUSY;
                //          __bic_SR_register_on_exit(LPM4_bits); //All Clocks and CPU etc
                    awake now that we received a byte. (check if it's last later)
                //If it is not the last Byte, the core will put us back into LPM0, which won
                    't stop the clocks, just the CPU
            break;
            case 0x01:
                if (P_RX_IN & S_ActiveSP.m_ucActiveSP_BitRx)
                    g_ucaRXBuffer[g_ucRXBufferIndex] |= 0x80;
                else
                    g_ucaRXBuffer[g_ucRXBufferIndex] &= ~0x80;
            break;
            default:
                if (P_RX_IN & S_ActiveSP.m_ucActiveSP_BitRx)
                    g_ucaRXBuffer[g_ucRXBufferIndex] |= 0x80;
                else
                    g_ucaRXBuffer[g_ucRXBufferIndex] &= ~0x80;
                g_ucaRXBuffer[g_ucRXBufferIndex] >>= 1;
            break;
        }
```

```
            g_ucRXBitsLeft--;
    }
}
```

A.4 irupt.c (Current Revision)

The following code demonstrates the current behavior of the Timer0 A0 interrupt service routine.

title

```
#pragma vector=TIMER0_A0_VECTOR
__interrupt void TIMER0_A0_ISR(void)
{
    if (g_ucCOMM_Flags & COMM_RUNNING)
    {
        __bic_SR_register_on_exit(LPM4_bits);
    }
}
```

B SP Code Modifications

B.1 comm.c (Current Revision)

This function contains the majority of the logic for byte transmission that was removed from the Timer A0 interrupt service routine.

SendByte Function

title

```
void vCOMM_SendByte(uint8 ucTXChar)
{
    uint8 ucParityBit;
    uint8 ucBitIdx;
    uint8 ucTXBitsLeft;

    // If we are already busy, return so as not to screw it up
    if (g_ucCOMM_Flags & COMM_TX_BUSY)
        return;

    // Indicate in the status register that we are now busy
    g_ucCOMM_Flags |=  COMM_TX_BUSY;
    P_RX_IE &= ~RX_PIN;

    // Calculate the parity bit prior to transmission
    ucParityBit = 0;
    for (ucBitIdx = 0; ucBitIdx < 8; ucBitIdx++)
    {
        ucParityBit ^= ((ucTXChar & 0x01) >> ucBitIdx);
    }

    // Reset the bit count so the ISR knows how many bits left to send
    ucTXBitsLeft = 0x0A;

    TA0CCR0 = g_unCOMM_BaudRateControl;

    // Starts the counter in 'Up-Mode'
    TACTL |= TACLR | MC_1;

    // Transmission loop which controls the UART timing for byte transmission
    while (g_ucCOMM_Flags & COMM_TX_BUSY)
    {
        switch (ucTXBitsLeft)
        {
            case 0x00:
                // Last bit is stop bit, return to idle state
                P_TX_OUT |= TX_PIN;
                g_ucCOMM_Flags &= ~COMM_TX_BUSY;
            break;

            case 0x01:
                if (ucParityBit)
                    P_TX_OUT |= TX_PIN;
                else
                    P_TX_OUT &= ~TX_PIN;
            break;

            case 0x0A:
                // First bit is start bit
                P_TX_OUT &= ~TX_PIN;
            break;

            default:
                // For data bits, mask to get correct value and the shift for next time
                if (ucTXChar & 0x01)
                    P_TX_OUT |= TX_PIN;
                else
                    P_TX_OUT &= ~TX_PIN;
                ucTXChar >>= 1;
            break;
        }

        __bis_SR_register(GIE + LPM0_bits);

        // Decrement the total bit count
        ucTXBitsLeft--;
    }

    P_RX_IE |= RX_PIN;

    // Stop the timer and show we are done
    TACTL &= ~(MC0 | MC1 | TAIFG);
}
```

receiveByte Function

title

```
////////////////////////////////////////////////////////////////////////////////
//! \brief Receives a byte via the software UART
//!
//!   \param none
//!   \return error code
//!   \sa TIMERA0_ISR(), vCOMM_Init()
////////////////////////////////////////////////////////////////////////////////
uint8 ucCOMM_ReceiveByte(void)
{
    uint8 ucRXBitsLeft;
    uint8 ucParityBit; // The calculated parity bit
    uint8 ucRxParityBit; // The received parity bit
    uint8 ucBitIdx;

    ucRXBitsLeft = 0x09;

    LPM0; //wait for start bit

    P_RX_IE &= ~RX_PIN; // Disable interrupts on the RX line
    TACTL = (TASSEL_2 | TACLR);
    TACCTL0 &= ~CCIE;

    // Start timer and delay for half a bit
    TACCR0 = g_unCOMM_BaudRateDelayControl;
    TACTL |= MC_1;
    while (!(TACTL & TAIFG));
    TACTL &= ~TAIFG;

    // Set up timer for comm. at the baud rate
    TACTL = (TASSEL_2 | TACLR);
    TACCTL0 = CCIE;
    TACCR0 = g_unCOMM_BaudRateControl;

    // Start the timer
    TACTL |= MC_1;

    while (g_ucCOMM_Flags & COMM_RX_BUSY)
    {
        switch (ucRXBitsLeft)
        {
            case 0x00:
                // There are no bits left, so lets reset all the values and stop timer
                TACTL &= ~(MC0 | MC1);
                P_RX_IE |= RX_PIN;
                P_RX_IFG &= ~RX_PIN;
                g_ucCOMM_Flags &= ~COMM_RX_BUSY;
            break;

                // Parity Bit
            case 0x01:
                P5OUT ^= BIT2;
                if (P_RX_IN & RX_PIN)
                    ucRxParityBit = 1;
                else
                    ucRxParityBit = 0;
                LPM0;
            break;

                // Last data bit no shift
            case 0x02:
                P5OUT ^= BIT2;
                if (P_RX_IN & RX_PIN)
                    g_ucaRXBuffer[g_ucRXBufferIndex] |= 0x80;
                else
                    g_ucaRXBuffer[g_ucRXBufferIndex] &= ~0x80;
                LPM0;
            break;

            default:
                P5OUT ^= BIT2;
                if (P_RX_IN & RX_PIN)
                    g_ucaRXBuffer[g_ucRXBufferIndex] |= 0x80;
                else
                    g_ucaRXBuffer[g_ucRXBufferIndex] &= ~0x80;
                g_ucaRXBuffer[g_ucRXBufferIndex] >>= 1;
                LPM0;
            break;

        }
        ucRXBitsLeft--;

    }

    g_ucRXBufferIndex++;    // Increment index for next byte

    // Check Parity
    ucParityBit = 0;
    for (ucBitIdx = 0; ucBitIdx < 8; ucBitIdx++)
    {
        ucParityBit ^= ((g_ucaRXBuffer[g_ucRXBufferIndex] & 0x01) >> ucBitIdx);
    }

    // ToDo move the parity check to the read from buffer function, leave the comm simple
//  if (ucParityBit != ucRxParityBit)
//      return COMM_PARITY_ERR;

    if (ucParityBit != ucRxParityBit)
        g_ucCOMM_Flags |= COMM_PARITY_ERR;

    return COMM_OK;

}
```

B.2 comm.c (Current Revision)

The following code provides the reduced Timer0 A0 interrupt service routine which handles the waking of the MCU from LPM0.

Timer0 A0 ISR

title

```
#pragma vector=TIMERA0_VECTOR
__interrupt void TIMERA0_ISR(void)
{

    if (g_ucCOMM_Flags & COMM_RUNNING)
    {
        __bic_SR_register_on_exit(LPM4_bits);
    }

}
```

B.3 comm.h (Previous Revision)

This following code demonstrates the hard-coded definitions for the baud rate delay values.

title

```
#define BAUD_115200_DELAY  0x000E//Slightly low to account for the set-up cycles that we
    delayed. At these times that's necessary...
//! \def BAUD_57600_DELAY
//! \brief Timer 0count for specific data rate delay, computed for 4Mhz SMCLK.
#define BAUD_57600_DELAY   0x0037
//! \def BAUD_19200_DELAY
//! \brief Timer count for specific data rate delay, computed for 4Mhz SMCLK.
#define BAUD_19200_DELAY   0x0068
//! \def BAUD_9600_DELAY
//! \brief Timer count for specific data rate delay, computed for 4Mhz SMCLK.
#define BAUD_9600_DELAY    0x00A3
//! \def BAUD_9600_DELAY
//! \brief Timer count for specific data rate delay, computed for 4Mhz SMCLK.
```

B.4 comm.h (Current Revision)

The following code demonstrates the current definitions for the baud rate delay values which utilizes the generalized delay cycle calculation formula.

title

```
//! \def BAUD_115200_DELAY
//! \brief Timer count for specific data rate delay, computed for 4Mhz SMCLK.
#define BAUD_115200_DELAY  BAUD_115200 + BAUD_115200/2 - 37 //0x0010//Slightly low to
    account for the set-up cycles that we delayed. At these times that's necessary...
//! \def BAUD_57600_DELAY
//! \brief Timer count for specific data rate delay, computed for 4Mhz SMCLK.
#define BAUD_57600_DELAY   BAUD_57600 + BAUD_57600/2 - 36
//! \def BAUD_19200_DELAY
//! \brief Timer count for specific data rate delay, computed for 4Mhz SMCLK.
#define BAUD_19200_DELAY   BAUD_19200 + BAUD_19200/2 - 36
//! \def BAUD_9600_DELAY
//! \brief Timer count for specific data rate delay, computed for 4Mhz SMCLK.
#define BAUD_9600_DELAY     BAUD_9600 + BAUD_9600/2 - 36
//! \def BAUD_9600_DELAY
//! \brief Timer count for specific data rate delay, computed for 4Mhz SMCLK.
```