
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	3
2 Code Reorganization	4
2.1 Interrupt Service Routines	4
2.2 Generalized Delay Solution	5
3 Faster Baud Rates	5
A CP Code Modifications	6
A.1 SP.c (Current Revision)	6
A.2 SP.h (Current Revision)	10
A.3 irupt.c (Previous Revision)	10
A.4 irupt.c (Current Revision)	12
B SP Code Modifications	13
B.1 comm.c (Current Revision)	13
B.2 comm.c (Current Revision)	16
B.3 comm.h (Previous Revision)	16
B.4 comm.h (Current Revision)	16

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).

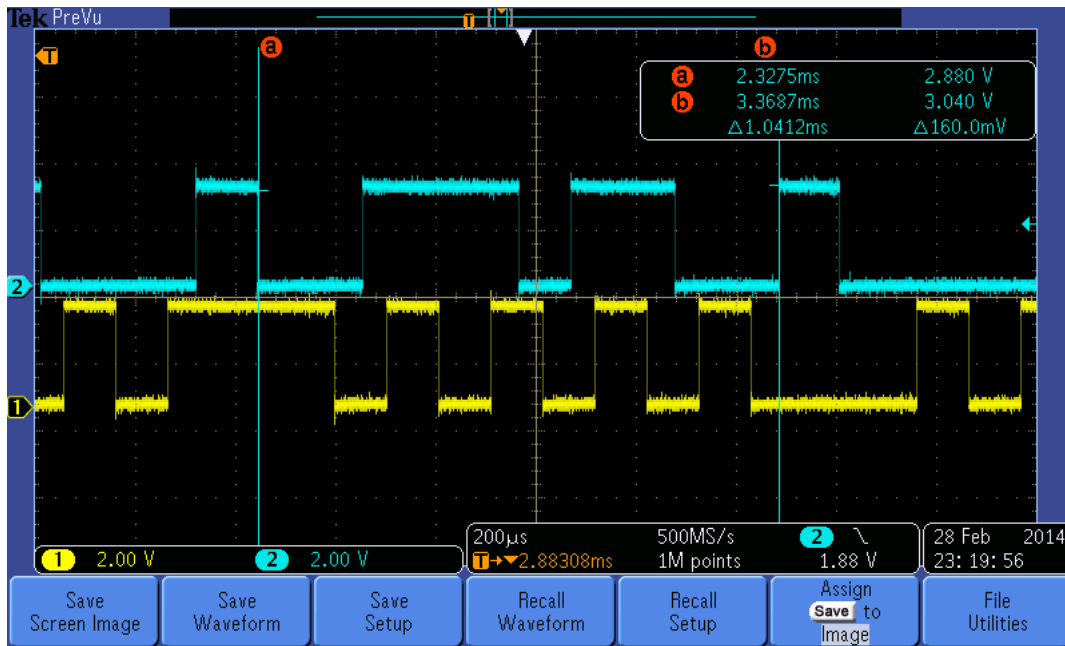


Figure 1: CP to SP byte transmission

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 1µs of the middle of every bit across all baud rates, whereas the code from the previous revision would often be within 2µs of the center of a bit.

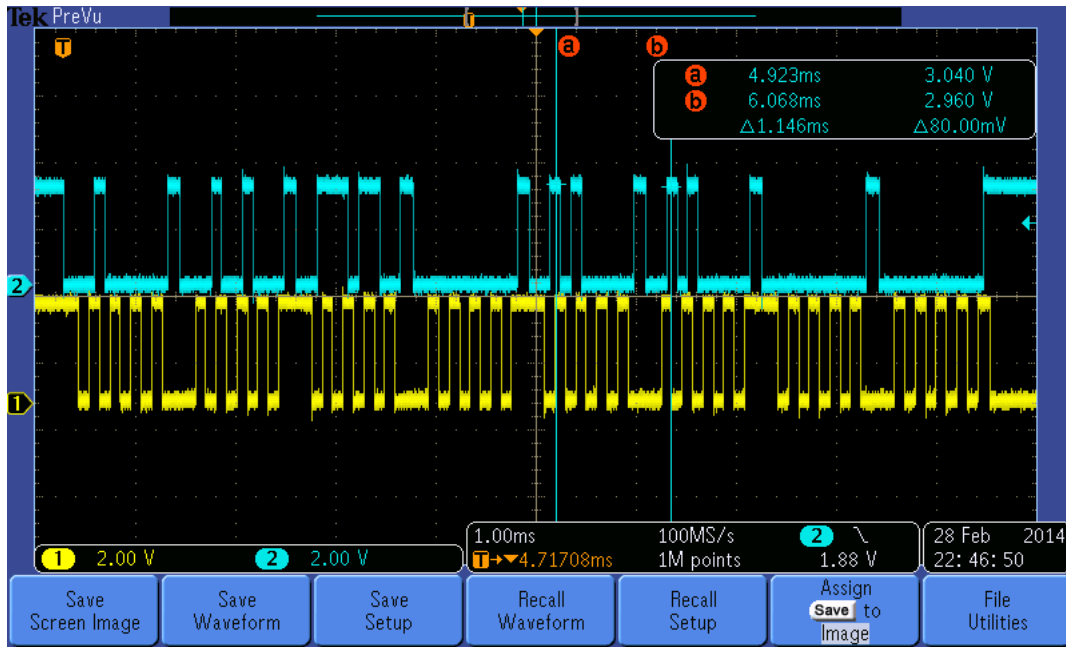


Figure 2: CP to SP 32-bit packet transmission

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 $[(1.5 * \text{BaudRateControl}) - 36]$ 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
560 //////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
561 //! \brief Sends a byte via the software UART
562 //!
563 //! This function pushes \e ucChar into the global TX buffer, where the
564 //! TimerA ISR can access it. The system drops into LPM0, keeping the SMCLK
565 //! alive for TimerA. The ISR handles the start and stop bits as well as the
566 //! baud rate. This is a blocking call and will not return until the software
567 //! has sent the entire message.
568 //! \param ucChar The 8-bit value to send
569 //! \return None
570 //////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
571 static void vSP_SendByte(uint8 ucTXChar)
572 {
573
574     uint8 ucParityBit;
575     uint8 ucBitIdx;
576     uint8 ucTXBitsLeft;
577
578     // If we are already busy, return so as not to screw it up
579     if (g_ucCOMM_Flags & COMMLTX_BUSY)
580         return;
581
582     P_RX_IE &= ~S_ActiveSP.m_ucActiveSP_BitRx;
583
584     // Indicate in the status register that we are now busy
585     g_ucCOMM_Flags |= COMMLTX_BUSY;
586
587     // Calculate the parity bit prior to transmission
588     ucParityBit = 0;
589     for(ucBitIdx = 0; ucBitIdx < 8; ucBitIdx++)
590     {
591         ucParityBit ^= ((ucTXChar & 0x01)>>ucBitIdx);
592     }
593
594     // Reset the bit count so the ISR knows how many bits left to send
595     ucTXBitsLeft = 0x0A;
596
597     TA0CCR0 = g_unCOMM_BaudRateControl;
598     // Starts the counter in 'Up-Mode'
599     TA0CTL |= TACLR | MC_1;
600
601     while (g_ucCOMM_Flags & COMMLTX_BUSY)
602     {
603         switch (ucTXBitsLeft)
604         {
605             case 0x00:
606                 // Last bit is stop bit, return to idle state
607                 *S_ActiveSP.m_ucActiveSP_RegOut |= S_ActiveSP.m_ucActiveSP_BitTx;
608                 __bis_SR_register(GIE + LPM0_bits);
609                 g_ucCOMM_Flags &= ~COMMLTX_BUSY;
610                 break;
611

```

```

612         case 0x01:
613             if (ucParityBit)
614                 *S_ActiveSP.m_ucActiveSP_RegOut |= S_ActiveSP.m_ucActiveSP_BitTx;
615             else
616                 *S_ActiveSP.m_ucActiveSP_RegOut &= ~S_ActiveSP.m_ucActiveSP_BitTx;
617
618             __bis_SR_register(GIE + LPM0_bits);
619             break;
620
621         case 0x0A:
622             if (ucTXChar & 0x01)
623                 *S_ActiveSP.m_ucActiveSP_RegOut &= ~S_ActiveSP.m_ucActiveSP_BitTx;
624             else
625                 *S_ActiveSP.m_ucActiveSP_RegOut &= ~S_ActiveSP.m_ucActiveSP_BitTx;
626
627             // First bit is start bit
628             // *S_ActiveSP.m_ucActiveSP_RegOut &= ~S_ActiveSP.m_ucActiveSP_BitTx;
629             __bis_SR_register(GIE + LPM0_bits);
630             break;
631
632         default:
633             // For data bits, mask to get correct value and the shift for next time
634             if (ucTXChar & 0x01)
635                 *S_ActiveSP.m_ucActiveSP_RegOut |= S_ActiveSP.m_ucActiveSP_BitTx;
636             else
637                 *S_ActiveSP.m_ucActiveSP_RegOut &= ~S_ActiveSP.m_ucActiveSP_BitTx;
638             ucTXChar >>= 1;
639             __bis_SR_register(GIE + LPM0_bits);
640             break;
641     }
642
643     // Decrement the total bit count
644     ucTXBitsLeft--;
645
646 }
647
648 P_RX_IE |= S_ActiveSP.m_ucActiveSP_BitRx;
649
650 // Stop the timer
651 TA0CTL &= ~(MC0 | MC1 | TAIFG);
652
653 P3OUT |= BIT4;
654 }
655
656 //////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////

```

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

ReceiveByte Function

```

title
656 //////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
657 //! \brief Receives a byte via the software UART
658 //!
659 //! \param None
660 //! \return None
661 //////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
662 static uint8 ucSP_ReceiveByte(void)
663 {
664     uint8 ucRXBitsLeft;
665     uint8 ucParityBit; // The calculated parity bit
666     uint8 ucRxParityBit; // The received parity bit

```

```

667     uint8 ucBitIdx;
668
669
670     // Set up timer
671     TA0CTL = (TASSEL_2 | TACLRL);
672     TA0CCTL0 &= ~CCIE;
673     TA0CCR0 = 0xFFFF;
674
675     // Wait for the port interrupt to exit LPM and continue
676     TA0CCTL0 |= CCIE;
677     TA0CTL |= MC_1;
678
679     // shut off MCU
680     LPM0;
681
682     P2IE &= ~0x0F; // Disable interrupts on the RX pin
683     TA0CTL = (TASSEL_2 | TACLRL);
684     TA0CCTL0 &= ~CCIE;
685
686     //If a port interrupt was received then we are in the RX active state else exit
687     if( !(g_ucCOMM_Flags & COMMRX_BUSY))
688     {
689         return COMMERROR;
690     }
691
692     ucRXBitsLeft = 0x09;
693
694     // Start timer and delay for half a bit
695     TA0CCR0 = g_unCOMM_BaudRateDelayControl;
696     TA0CTL |= MC_1;
697     while (!(TA0CTL & TAIFG));
698     TA0CTL &= ~TAIFG;
699
700     // Set up timer for comm. at the baud rate
701     TA0CTL = (TASSEL_2 | TACLRL);
702     TA0CCTL0 = CCIE;
703     TA0CCR0 = g_unCOMM_BaudRateControl;
704
705     // Start the timer
706     TA0CTL |= MC_1;
707
708     while(g_ucCOMM_Flags & COMMRX_BUSY)
709     {
710         switch (ucRXBitsLeft)
711         {
712             case 0x00:
713                 // There are no bits left, so lets reset all the values and stop timer
714                 TA0CTL = (TASSEL_2 | TACLRL);
715                 P_RX_IFG &= ~S_ActiveSP.m_ucActiveSP_BitRx;
716                 P_RX_IE |= S_ActiveSP.m_ucActiveSP_BitRx;
717                 g_ucCOMM_Flags &= ~COMMRX_BUSY;
718                 // Increment index for next byte
719                 g_ucRXBufferIndex++;
720                 break;
721
722                 // Parity Bit
723             case 0x01:
724                 if (P_RX_IN & S_ActiveSP.m_ucActiveSP_BitRx)
725                     ucRxParityBit = 1;
726                 else
727                     ucRxParityBit = 0;
728

```



```

729         // shut off MCU
730         LPM0;
731         break;
732
733         // Last data bit no shift
734         case 0x02:
735             if (P_RX_IN & S_ActiveSP.m_ucActiveSP_BitRx)
736                 g_ucRXBuffer[g_ucRXBufferIndex] |= 0x80;
737             else
738                 g_ucRXBuffer[g_ucRXBufferIndex] &= ~0x80;
739
740             // shut off MCU
741             LPM0;
742             break;
743
744         default:
745             if (P_RX_IN & S_ActiveSP.m_ucActiveSP_BitRx)
746                 g_ucRXBuffer[g_ucRXBufferIndex] |= 0x80;
747             else
748                 g_ucRXBuffer[g_ucRXBufferIndex] &= ~0x80;
749
750             g_ucRXBuffer[g_ucRXBufferIndex] >>= 1;
751             LPM0;
752             break;
753
754     }
755     ucRXBitsLeft--;
756
757 }
758
759 // Check Parity
760 // ucParityBit = 0;
761 // for(ucBitIdx = 0; ucBitIdx < 8; ucBitIdx++)
762 // {
763 //     ucParityBit ^= ((g_ucRXBuffer[g_ucRXBufferIndex] & 0x01)>>ucBitIdx);
764 // }
765
766 //Todo move the parity check to the read from buffer function to keep comm simple
767 // if(ucParityBit != ucRxParityBit)
768 //     return COMMParity_ERR;
769
770 return COMMOK;
771 }
772
773 ///////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
774 //! \brief Shuts off the software modules
775 //!
776 //! This shuts down TimerA and disables all of the interrupts used. The vSP_ShutdownComm()
777 //! function must be used when switching between SP boards otherwise the communication
778 //! 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
838 uint8 ucSP_WaitForMessage(void)
839 {
840
841     // Set the size of the received message to the minimum
842     g_ucRXMessageSize = SP_HEADERSIZE;
843

```

```

844 // Wait to receive the message
845 do // saves 3 cycles as opposed to while
846 {
847
848     if(ucSP_ReceiveByte())
849     {
850         return COMMERROR;
851     }
852
853     // If we have received the header of the message, update the RX message to
854     // the size of the message received
855     if (g_ucRXBufferIndex == SP_HEADERSIZE)
856     {
857         g_ucRXMessageSize = g_ucaRXBuffer[0x01];
858
859         // Range check the g_ucRXMessageSize variable
860         if (g_ucRXMessageSize > MAX_SP_MSGSIZE || g_ucRXMessageSize < SP_HEADERSIZE)
861             return 0x04;
862     }
863
864 } while (g_ucRXBufferIndex != g_ucRXMessageSize);
865
866 // No message received
867 if (g_ucRXBufferIndex == 0)
868 {
869     g_ucRXMessageSize = 0;
870     return COMMERROR;
871 }
872
873 //success
874 return 0;
875 }

```

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
349 //! \def BAUD_115200_DELAY
350 //! \brief Timer count for specific data rate delay, computed for 4Mhz SMCLK.
351 #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...
352 //! \def BAUD_57600_DELAY
353 //! \brief Timer count for specific data rate delay, computed for 4Mhz SMCLK.
354 #define BAUD_57600_DELAY BAUD_57600 + BAUD_57600/2 - 36
355 //! \def BAUD_19200_DELAY
356 //! \brief Timer count for specific data rate delay, computed for 4Mhz SMCLK.
357 #define BAUD_19200_DELAY BAUD_19200 + BAUD_19200/2 - 36
358 //! \def BAUD_9600_DELAY
359 //! \brief Timer count for specific data rate delay, computed for 4Mhz SMCLK.
360 #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
250 #pragma vector=TIMER0_A0_VECTOR

```

```

251 __interrupt void TIMER0_A0_ISR(void)
252 {
253
254     if (g_ucCOMM_Flags & COMMLTX_BUSY)
255     {
256         switch (g_ucTXBitsLeft)
257         {
258             case 0x00:
259                 // If there are no bits left, then return to function call
260                 __bic_SR_register_on_exit(LPM0_bits);
261                 break;
262             case 0x01:
263                 // Last bit is stop bit, return to idle state
264                 *S_ActiveSP.m_ucActiveSP_RegOut |= S_ActiveSP.m_ucActiveSP_BitTx;
265                 __bic_SR_register_on_exit(LPM0_bits);
266                 break;
267             case 0x0A:
268                 // First bit is start bit
269                 *S_ActiveSP.m_ucActiveSP_RegOut &= ~S_ActiveSP.m_ucActiveSP_BitTx;
270                 break;
271             default:
272                 // For data bits, mask to get correct value and the shift for next time
273                 if (g_ucTXBuffer & 0x01)
274                     *S_ActiveSP.m_ucActiveSP_RegOut |= S_ActiveSP.m_ucActiveSP_BitTx;
275                 else
276                     *S_ActiveSP.m_ucActiveSP_RegOut &= ~S_ActiveSP.m_ucActiveSP_BitTx;
277                 g_ucTXBuffer >>= 1;
278                 break;
279         }
280         // Decrement the total bit count
281         g_ucTXBitsLeft--;
282     }
283     if (g_ucCOMM_Flags & COMMRX_BUSY)
284     {
285         switch (g_ucRXBitsLeft)
286         {
287             case 0x00:
288                 // There are no bits left, so lets reset all the values and stop timer
289                 TA0CTL &= ~(MC0 | MC1);
290                 P_RX_IE |= S_ActiveSP.m_ucActiveSP_BitRx;
291                 P_RX_IFG &= ~S_ActiveSP.m_ucActiveSP_BitRx;
292                 g_ucRXBufferIndex++;
293                 g_ucCOMM_Flags &= ~COMMRX_BUSY;
294                 // __bic_SR_register_on_exit(LPM4_bits); //All Clocks and CPU etc
295                 // awake now that we received a byte. (check if it's last later)
296                 // If it is not the last Byte, the core will put us back into LPM0, which won
297                 // 't stop the clocks, just the CPU
298                 break;
299             case 0x01:
300                 if (P_RX_IN & S_ActiveSP.m_ucActiveSP_BitRx)
301                     g_ucRXBuffer[g_ucRXBufferIndex] |= 0x80;
302                 else
303                     g_ucRXBuffer[g_ucRXBufferIndex] &= ~0x80;
304                 break;
305             default:
306                 if (P_RX_IN & S_ActiveSP.m_ucActiveSP_BitRx)
307                     g_ucRXBuffer[g_ucRXBufferIndex] |= 0x80;
308                 else
309                     g_ucRXBuffer[g_ucRXBufferIndex] &= ~0x80;
310                 g_ucRXBuffer[g_ucRXBufferIndex] >>= 1;
311                 break;
312         }
313     }

```

```
311         g_ucRXBitsLeft--;
312     }
313 }
```

A.4 *irupt.c* (Current Revision)

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

```
title
237 #pragma vector=TIMER0_A0_VECTOR
238 __interrupt void TIMER0_A0_ISR(void)
239 {
240     if (g_ucCOMM_Flags & COMMRUNNING)
241     {
242         __bic_SR_register_on_exit(LPM4_bits);
243     }
244 }
```

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
227 void vCOMM_SendByte(uint8 ucTXChar)
228 {
229     uint8 ucParityBit;
230     uint8 ucBitIdx;
231     uint8 ucTXBitsLeft;
232
233     // If we are already busy, return so as not to screw it up
234     if (g_ucCOMM_Flags & COMMLTX_BUSY)
235         return;
236
237     // Indicate in the status register that we are now busy
238     g_ucCOMM_Flags |= COMMLTX_BUSY;
239     P_RX_IE &= ~RX_PIN;
240
241     // Calculate the parity bit prior to transmission
242     ucParityBit = 0;
243     for (ucBitIdx = 0; ucBitIdx < 8; ucBitIdx++)
244     {
245         ucParityBit ^= ((ucTXChar & 0x01) >> ucBitIdx);
246     }
247
248     // Reset the bit count so the ISR knows how many bits left to send
249     ucTXBitsLeft = 0x0A;
250
251     TA0CCR0 = g_unCOMM_BaudRateControl;
252
253     // Starts the counter in 'Up-Mode'
254     TACTL |= TACLRL | MC_1;
255
256     // Transmission loop which controls the UART timing for byte transmission
257     while (g_ucCOMM_Flags & COMMLTX_BUSY)
258     {
259         switch (ucTXBitsLeft)
260         {
261             case 0x00:
262                 // Last bit is stop bit, return to idle state
263                 P_TX_OUT |= TX_PIN;
264                 g_ucCOMM_Flags &= ~COMMLTX_BUSY;
265                 break;
266
267             case 0x01:
268                 if (ucParityBit)
269                     P_TX_OUT |= TX_PIN;
270                 else
271                     P_TX_OUT &= ~TX_PIN;
272                 break;
273
274             case 0x0A:
275                 // First bit is start bit
276                 P_TX_OUT &= ~TX_PIN;
277                 break;
278

```

```

279         default:
280             // For data bits, mask to get correct value and the shift for next time
281             if (ucTXChar & 0x01)
282                 P_TX_OUT |= TX_PIN;
283             else
284                 P_TX_OUT &= ~TX_PIN;
285             ucTXChar >>= 1;
286             break;
287     }
288
289     __bis_SR_register(GIE + LPM0_bits);
290
291     // Decrement the total bit count
292     ucTXBitsLeft--;
293 }
294
295 P_RX_IE |= RX_PIN;
296
297 // Stop the timer and show we are done
298 TACTL &= ~(MC0 | MC1 | TAIFG);
299 }

```

receiveByte Function

```

title
301 ///////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
302 //! \brief Receives a byte via the software UART
303 //!
304 //! \param none
305 //! \return error code
306 //! \sa TIMERA0_ISR(), vCOMM_Init()
307 ///////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
308 uint8 ucCOMM_ReceiveByte(void)
309 {
310     uint8 ucRXBitsLeft;
311     uint8 ucParityBit; // The calculated parity bit
312     uint8 ucRxParityBit; // The received parity bit
313     uint8 ucBitIdx;
314
315     ucRXBitsLeft = 0x09;
316
317     LPM0; //wait for start bit
318
319     P_RX_IE &= ~RX_PIN; // Disable interrupts on the RX line
320     TACTL = (TASSEL_2 | TACL_R);
321     TACCTL0 &= ~CCIE;
322
323     // Start timer and delay for half a bit
324     TACCR0 = g_unCOMM_BaudRateDelayControl;
325     TACTL |= MC_1;
326     while (!(TACTL & TAIFG));
327     TACTL &= ~TAIFG;
328
329     // Set up timer for comm. at the baud rate
330     TACTL = (TASSEL_2 | TACL_R);
331     TACCTL0 = CCIE;
332     TACCR0 = g_unCOMM_BaudRateControl;
333
334     // Start the timer
335     TACTL |= MC_1;
336
337     while (g_ucCOMM_Flags & COMMRX_BUSY)

```

```

338     {
339         switch (ucRXBitsLeft)
340         {
341             case 0x00:
342                 // There are no bits left , so lets reset all the values and stop timer
343                 TACTL &= ~(MC0 | MC1);
344                 P_RX_IE |= RX_PIN;
345                 P_RX_IFG &= ~RX_PIN;
346                 g_ucCOMM_Flags &= ~COMMRX_BUSY;
347                 break;
348
349                 // Parity Bit
350             case 0x01:
351                 P5OUT ^= BIT2;
352                 if (P_RX_IN & RX_PIN)
353                     ucRxParityBit = 1;
354                 else
355                     ucRxParityBit = 0;
356                 LPM0;
357                 break;
358
359                 // Last data bit no shift
360             case 0x02:
361                 P5OUT ^= BIT2;
362                 if (P_RX_IN & RX_PIN)
363                     g_ucaRXBuffer[g_ucRXBufferIndex] |= 0x80;
364                 else
365                     g_ucaRXBuffer[g_ucRXBufferIndex] &= ~0x80;
366                 LPM0;
367                 break;
368
369             default:
370                 P5OUT ^= BIT2;
371                 if (P_RX_IN & RX_PIN)
372                     g_ucaRXBuffer[g_ucRXBufferIndex] |= 0x80;
373                 else
374                     g_ucaRXBuffer[g_ucRXBufferIndex] &= ~0x80;
375                 g_ucaRXBuffer[g_ucRXBufferIndex] >>= 1;
376                 LPM0;
377                 break;
378         }
379         ucRXBitsLeft--;
380     }
381
382 }
383
384 g_ucRXBufferIndex++;    // Increment index for next byte
385
386 // Check Parity
387 ucParityBit = 0;
388 for (ucBitIdx = 0; ucBitIdx < 8; ucBitIdx++)
389 {
390     ucParityBit ^= ((g_ucaRXBuffer[g_ucRXBufferIndex] & 0x01) >> ucBitIdx);
391 }
392
393 // ToDo move the parity check to the read from buffer function , leave the comm simple
394 // if (ucParityBit != ucRxParityBit)
395 //     return COMM_PARITY_ERR;
396
397 if (ucParityBit != ucRxParityBit)
398     g_ucCOMM_Flags |= COMM_PARITY_ERR;
399

```

```

400     return COMMOK;
401
402 }

```

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
683 #pragma vector=TIMERA0_VECTOR
684 __interrupt void TIMERA0_ISR(void)
685 {
686
687     if (g_ucCOMM_Flags & COMMRUNNING)
688     {
689         __bic_SR_register_on_exit(LPM4_bits);
690     }
691
692 }

```

B.3 *comm.h* (Previous Revision)

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

```

title
96 #define BAUD_115200_DELAY 0x000E // Slightly low to account for the set-up cycles that we
    delayed. At these times that's necessary...
97 //! \def BAUD_57600_DELAY
98 //! \brief Timer 0 count for specific data rate delay, computed for 4Mhz SMCLK.
99 #define BAUD_57600_DELAY 0x0037
100 //! \def BAUD_19200_DELAY
101 //! \brief Timer count for specific data rate delay, computed for 4Mhz SMCLK.
102 #define BAUD_19200_DELAY 0x0068
103 //! \def BAUD_9600_DELAY
104 //! \brief Timer count for specific data rate delay, computed for 4Mhz SMCLK.
105 #define BAUD_9600_DELAY 0x00A3
106 //! \def BAUD_9600_DELAY
107 //! \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
97 //! \def BAUD_115200_DELAY
98 //! \brief Timer count for specific data rate delay, computed for 4Mhz SMCLK.
99 #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...
100 //! \def BAUD_57600_DELAY
101 //! \brief Timer count for specific data rate delay, computed for 4Mhz SMCLK.
102 #define BAUD_57600_DELAY BAUD_57600 + BAUD_57600/2 - 36
103 //! \def BAUD_19200_DELAY
104 //! \brief Timer count for specific data rate delay, computed for 4Mhz SMCLK.
105 #define BAUD_19200_DELAY BAUD_19200 + BAUD_19200/2 - 36
106 //! \def BAUD_9600_DELAY

```



```
107 //! \brief Timer count for specific data rate delay, computed for 4Mhz SMCLK.
108 #define BAUD_9600_DELAY    BAUD_9600 + BAUD_9600/2 - 36
109 //! \def BAUD_9600_DELAY
110 //! \brief Timer count for specific data rate delay, computed for 4Mhz SMCLK.
```