Application Report
SLUA782 – May 2017

UART-Based Communication Protocol for Inter-module Communication of Light Vehicle Systems

Steven Yao *BMS: High Cell & Emerging*

ABSTRACT

This application report describes a protocol based on a universal asynchronous receiver/transmitter (UART) for E-Bike for LEV battery communication with an external host. For better anti-interference performance, the communication interface adopts the differential CAN bus as the physical layer. For energy saving purposes, a wake-up mechanism is implemented to wake up the device. A concise and easy protocol is implemented based on the *Idle-line Multiprocessor Format* of MSP430x2xx UART serial interface.

Contents

List of Figures

1 Introduction

Li-ion batteries continue to become more widely used in electrically powered light vehicles such as E-Bikes, wheel chairs, and scooters due to their high energy density and long cycle life. Meanwhile, a smarter and more intelligent battery management system (BMS) is also required for better user experience and easier maintenance services. A digital interface must be implemented to make the system smarter.

TI's bq78350 battery management IC provides a full range of management to Li-ion or Li-Fe batteries including protection, gauging, balancing, and lifetime data logging. The bq78350 greatly reduces the development time and additionally provides SMBus as do other congeneric battery management ICs from TI. Such interfaces are typically used on the same PCB along with each individual IC. For module-to-module or board-level communication, it is not suitable due to its susceptibility to noise interference.

In this application note, a half-duplex interface with a differential pair of CAN physical layers is used. For cost-saving purposes, the CAN protocol is not used, saving the cost for a CAN controller, thus only a UART engine instead of a CAN controller integrated in the MSP430 MCU, as well as a CAN transceiver like the ISO1050 are needed.

With these considerations, a system with the best performance and cost ratio can be achieved with the bq76920/30/40 + bq78350 + MSP430G2553 + ISO1050. Figure 1 is the hardware setup of the total solution using TI EVMs.

2 Hardware Setup

The total solution can be setup with existing TI EVMs, with the exception of wake up, shutdown control, and the level-shifter circuit, as depicted in Figure 1.

2.1 *bq76920, bq76930, and bq76940EVM*

Three different EVMs can be selected and connected to a battery group: bq76920EVM, bq76930EVM, and bq76940EVM. Select the EVM depending on the actual number of cells connected in-series in the battery group:

- Use bq76920EVM for 3s to 5s cells in serial
- Use bq76930EVM for 6s to 10s cells in serial
- Use bq76940EVM for 9s to 15s cells in serial

In this application report, bq769x0EVM is used as an appellative name or for all 3 types of EVM (bq76920/30/40EVM) hereafter. All these EVMs can be configured to AFE mode or gauge mode, the system described in this application report needs to configure the EVM to work under gauge mode. To configure the EVM to gauge mode, refer to section 2.2.2 in the bq76930EVM and bq76940EVM user guide (SLVU925), or section 2.2.2 in the bq76920EVM user guide (SLVU924).

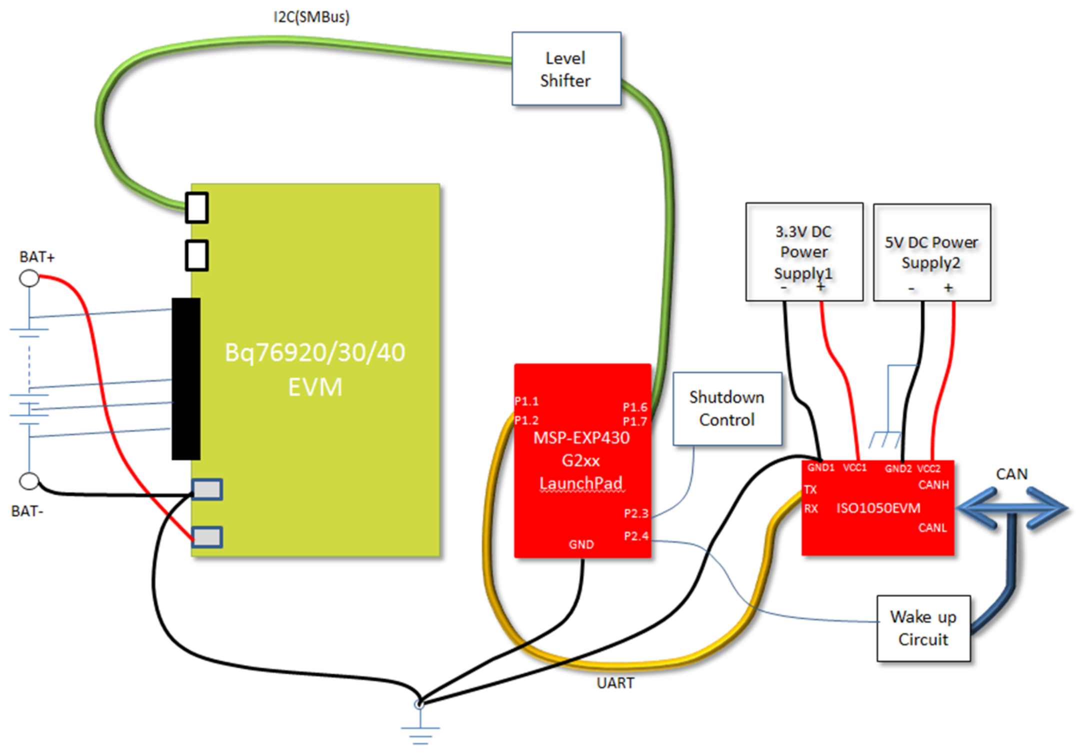


Figure 1. Hardware Setup

2.2 MSP-EXP430G2 LaunchPad™

For the MSP-EXP430G2 LaunchPad™ configuration, refer to the *MSP-EXP430G2 LaunchPad Development Kit* user guide (SLAU318). A 20-pin MSP430G2553 IC in a DIP package can be installed on this LaunchPad for evaluation. For MCU configuration, P1.1 and P1.2 are configured as UART interface, and P1.6, P1.7 are configured as an I^{2}C interface. P2.3 is configured as an input pin to detect the signal from the shutdown control circuit. P2.4 is configured as an input pin to detect the signal from the wake up circuit.

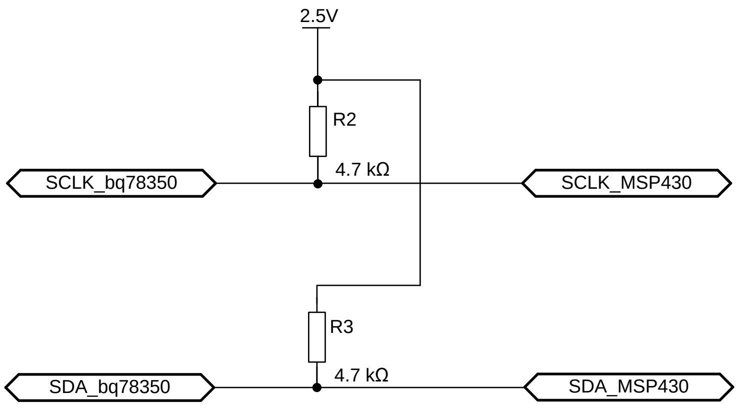
When debugging the program, the LaunchPad is connected to the PC via a USB cable, the power is provided by the PC. The target MCU (MSP430G2553) section can be totally disconnected with the JTAG section by removing all 5 jumpers on J3 of the LaunchPad. The VCC and GND of the target MCU (MSP430G2553) section can be connected to "+" and "–" output of 3.3-V DC *Power Supply1*.

2.3 ISO1050EVM

The ISO1050EVM can be powered with 2 independent DC power supplies with isolated GND. *Power Supply1* can be configured to 3.3 V so the UART interface can work under the same voltage range with the LaunchPad. *Power Supply2* is used for the CAN bus side and can be configured to 5 V so that the dynamic range is enough for better noise immunity.

2.4 Level Shifter Circuit

A level shifter is needed to bridge the communication between the MSP430G2553 on the LaunchPad and the bq78350 on the bq769x0EVM as this two devices are working under different voltage range, the typical level shifter can be implemented with the schematic in Figure 2.



Copyright © 2016, Texas Instruments Incorporated

Figure 2. Level-Shifter Schematic (MSP430G2553 and bq78350)

Since P1.6 and P1.7 of the MSP430 are configured to an I^{2}C port, they are in open drain status for logic "1" output, only a 2.5-V pull-up voltage (from REGOUT of bq769x0) is required for communicating with the bq78350, no additional components are needed for level transition although the MSP430 works under 3.3-V power supply.

2.5 Shutdown Control Circuit

In a real application, customers always request the system to transit to shutdown mode to save battery energy; a switch button can be used for this purpose. The related circuit is shown in Figure 3.

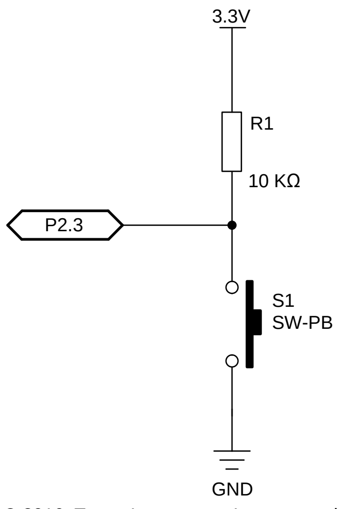


Figure 3. Shutdown Control Circuit

3.3 V is the power supply of the MSP430. Pushing down S1 generates a high-to-low transition on P2.3 of the MSP430. This pin should be configured to trigger an interrupt at such a transition.

2.6 Wake-Up Circuit

The system can transit to *Sleep Mode* from normal mode if there is no communication detected on the CAN bus for 20 seconds. When transiting to *Sleep Mode*, the MCU sends a SLEEP MAC command to the bq78350 via the SMBus, and then sends itself to LPM3 mode to save power. To wake up the device from *Sleep Mode*, according to the schematic in Figure 4, a frame with any information initiated by host on the CAN bus is required, such frame is not be handled by the MCU, but generates a high-to-low transition on P2.4, which should be configured to trigger an interrupt when the transition is detected. The device returns from *Sleep Mode* to normal mode in the corresponding interrupt routine. The interrupt of P2.4 should be also disabled in the interrupt routine to exit from the low power mode and enabled until the next time, right before the MCU decides to go to *Sleep Mode*.

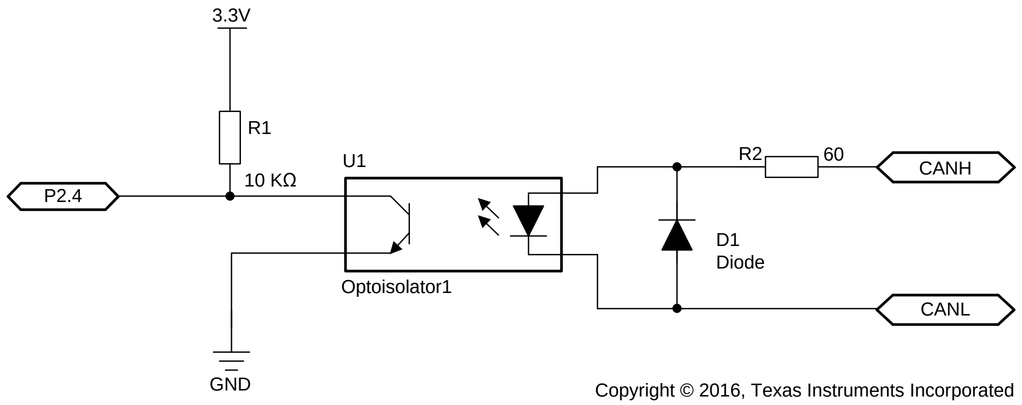


Figure 4. Wake-Up Circuit

3 System Working Modes Description

The system works under 3 modes: *Normal Mode*, *Sleep Mode* and *Shutdown Mode*.

3.1 Shutdown Mode

When the battery group (left-most side in Figure 1) is connected to the bq769x0EVM, the system is in *Shutdown Mode*, at this moment, the 2.5V LDO output of bq769x0 is off, bq78350 is also in power off state. The 3.3-V power supply for the MCU does not affect the bq769x0EVM side. In a real implementation, the MCU can share the LDO from the bq769x0. If this is the case, the MCU is also powered off. And the power for the ISO1050 should be controlled by MCU, when MCU is powered off, the DC/DC providing power to ISO1050 should be also designed in power off state. Thus the entire system consumes very little current.

3.2 Normal Mode

To activate the system, a voltage above 1 V should be applied to the TS1 pin of the bq769x0 to activate bq769x0, then a 2.5-V output presents on the LDO output of the bq769x0, the gauge IC bq78350 is powered up accordingly. For details and configuration information on the bq78350, see the bq78350-R1 datasheet (SLUSCD0) and technical reference manual (TRM) (SLUUBD3). A properly configured bq78350 fulfills functions like battery protection against fault conditions such as OV, UV, OC, SC, OT, UT, and so forth. The bq78350 accomplishes cell balancing according the preset balancing conditions and gauges the battery capacity with the TI-patented CEDV algorithm. Meanwhile, it responds to any valid host access by returning the battery information like battery voltage, cell voltages, pack current, temperature, remaining capacity, SOC as expected via the SMBus.

With 2.5-V output available, the MSP430 MCU reads the any information once it is requested, according to the command received from the UART interface. The MCU can also instruct the bq78350 to execute any commands defined in the TRM, including shutting down the bq769x0 and bq78350.

3.3 Sleep Mode

A sleep timer is implemented in the MSP430 MCU firmware. When the system is in *Normal Mode*, and the MCU has not received any command on the UART interface for a defined time period (the default is 20 seconds) the sleep timer counts up to 20 seconds, and the MCU instructs the bq78350 to enter *Sleep Mode*, and sends itself to LPM3 low power mode, with the consumption current less than 1 µA. When the device is in *Sleep Mode*, any frame on the CAN bus side of the ISO1050 can wake it up, the frame wakes up the MCU, is abandoned by MCU, the MCU then wakes up the system and prepares to receive any data over the UART interface 1 second later, then responds to or executes any command if the address matches with the predefined address in the UART engine. This means that when the MCU is in *Sleep Mode*, the first frame targeted to it is not received by the MCU. The external host has to send it again after 1 second, if it is expected to be correctly received by the MCU in the battery.

4 Protocol Description

4.1 Idle Line Multiprocessor Format

For future support of E-Bikes or E-Motor architecture, an idle-line multiprocessor format for the MSP430 UART engine is adopted. This mode is suitable for low-power applications like a battery management system. In this mode, a specific address can be given to each unit attached to the same physical interface. Each unit only responds to the access with its matching address. Only the host unit of the vehicle would initiate the accesses to other slave units. With such UART format, the physical layer of the CAN bus can be utilized for better noise rejection, therefore, low *Bit Error Rate*. With such mechanism, a low-cost yet relatively reliable communication interface can be implemented in light vehicle systems. For more details about *Idle Line Multiprocessor* format, see section 15.3.3 in the *MSP430x2xx Family User's Guide* (SLAU144).

4.2 Host to Slave Frame Definition

For a transaction initiated by the host, it is always preceded with a frame from the host to the target slave unit. The frame structure can be defined as shown in Table 1.

Table 1. Host to Slave Frame Definition

	Byte 1	Byte 2	Byte 3	Byte 4	Byte 5	…	Byte n	Byte n+1	Byte n+2
Write Command	Address(0x4a)	FrameLength (n–2)	Operation Type(1,write)	Command Byte 1	Command Byte 2	…	Command Byte (n–3)	CheckSum	
Read Command	Address(0x4a)	FrameLength (n–1)	Operation Type(2,read)	Command Byte 1	Command Byte 2	…	Command Byte (n–3)	Number of Bytes to be read (≤ 32)	CheckSum
Execution Command	Address(0x4a)	FrameLength (n–2)	Operation Type(3,Execution)	Command Byte 1	Command Byte 2	…	Command Byte (n–3)	CheckSum	

There are 3 types of frame for the host to initiate an access to the target unit; (1) *Write Command* frame, (2) *Read Command* frame, and (3) *Execution Command* frame.

The *Write Command* sends data to the target unit, no return data or any immediate action of the target unit is expected. Such commands can be used to transmit data from the host to the slave unit. The final objective of the transmission in this application can be either MCU or bq78350, the Byte 3 in the frame can be a register to hold the following data from Byte 4 to Byte n, if the bq78350 register X is used here as Byte 3, the following bytes from Byte 4 to Byte n will be sent to register X of bq78350. For example, given the target address is 0x4a, and the host is trying to read the firmware version of bq78350, in precedence, the host should send a *Write Command* frame like this: 0x4a, 0x04, 0x02, 0x00, 0x02, 0x00, 0x08. The 1st byte 0x4a is the address of the unit; here the unit refers to the entire battery system. The 2nd byte is the length of the frame with the last byte for Checksum and first 2 bytes for unit address and frame length itself excluded. The 3rd byte is the operation type of the command, 0x02 indicates a *Write Command* frame. The 4th byte is the register address; here 0x00 is the bq78350 MAC command register. The 5th and 6th bytes 0x02 and 0x00 instruct the bq78350 to prepare the words representing the bq78350 firmware version at another register for the host to read back. The last byte is the Checksum byte of the frame. For the register addresses, the register address for the bq78350 can be directly used here, and the left unused 8-bits code can be used as the register address for future function development of the MSP430 MCU.

The *Read Command* instructs the unit to send the expected data back to the host. For the *Write Command* previously discussed, after the command is received by the MCU, the MCU interprets this command and sends the appropriate command to the bq78350 over the SMBus between the MCU and bq78350, after the bq78350 receives the command, which is for reading the firmware version number in this example, it then prepares the data in a specific register. For the host to get this data, it needs to send another command, which is to have the MCU to read the data over the SMBus and send it back to the host via UART and CAN interface. The *Read Command* for the host to send is like this: 0x4a, 0x03, 0x01, 0x23, 0x05, and 0x2c. The 1st byte is the unit address of the battery unit. The 2nd byte is the frame length with the last byte and first two bytes excluded. The 3rd byte, 0x01, indicates this frame is for read operation. The 4th byte is the address of the register of the bq78350 with the returned data prepared. The 5th byte is the expected number of bytes to be read back from the bq78350. The last byte is the Checksum byte.

The *Execution Command* frame is for the host to instruct the MCU to execute a desired operation. The *Operation Type* code is 0x03, at present, the code for possible operations has not been defined.

4.3 Slave to Host Frame Definition

In response to the host access, the slave unit sends the frame to the host with the frames defined in Table 2.

If the slave unit sees the Checksum is incorrect, it sends two successive 0xFF bytes which is defined in Table 2 as Checksum Error frame to indicate to the host that the frame might be corrupted by noise interference somehow, the host typically should send the previous frame with the correct Checksum byte again.

Table 2. Slave to Host Frame Definition

	Byte 1	Byte 2	Byte 3	…	Byte n+1	Byten+2
CheckSum Error	0xFF	0xFF				
bq78350 I2C Error	0xFE	0xFE				
UART Timeout Error	0xFD	0xFD				
Wake Up Indication	0xFC	0xFC				
Execution Command Fail	0xFB	0xFB				
Execution Command Done	0x00	0x00				
Response With Additional Bytes	Length of the bytes following n(1–32)	Response byte 1	Response byte 2	…	Response byte n	Checksum

The second type of frame is the bq78350 I2C Error. This frame indicates that the MCU has not been able to communicate with the bq78350 successfully in the latest SMBus access; there might be something wrong with the bq78350 inside the battery.

UART Timeout Error frame indicates that the frame received by the slave unit is incomplete, or a completed frame from the host has not been received within the expected time period.

The *Execution Command Done Frame* indicates to the host that the immediate previous *Execution Command* has been completed successfully.

The *Execution Command Fail Frame* indicates to the host that the immediate previous *Execution Command* has not been completed successfully.

Response With Additional Bytes Frame returns a string to the host, the first byte is the length of the string, the range given here is from 1–32, however, it can be larger if the MCU has enough memory to be used as buffer. But it is not suggested to be larger than 0x40, as a frame with the first byte between 0x40 to 0x7F is reserved for the address of the slave devices in a system such as an E-Bike, the numbers above 0x7F can be used as status or error indicators to the host.

4.4 Checksum

The cable from the slave unit (Battery) to the host can be 1 meter long and susceptible to noise interference in a real application. For better noise immunity and lower bit error rate, a Checksum byte is used for both the frames sent from the host to the slave units and the opposite. For the frames from the host to the slave unit, the checksum value is the result of the sum of the bytes from the 2nd byte to the byte preceding the Checksum byte in the frame modulo 256. Once the MCU found the Checksum calculated by itself does not match the Checksum byte sent by the host, it sends a *Checksum Error* frame to the host. For the frames from the slave unit to the host, the Checksum byte is the result of the sum of all the bytes in precedence to the Checksum byte modulo 256. The host calculates the checksum and compares the value to the Checksum byte it received at the end of each frame, if it mismatches, the host discards the frame.

5 Code Implementation

The code basically consists of 3 parts. When the system is activated, the MCU starts the firmware by initializing various internal modules, then enabled interrupts and enters the main loop of the program.

5.1 Initialize

The first module need to be initialized is the *Clock Module*. In this application, the Clock is initialized to 16 MHz, and the internal 12-kHz VLO is used as the frequency of ACLK. The clock initialization code follows:

```
void ClockInitialise()
{
     DCOCTL = CALDCO_16MHZ;
     BCSCTL1 |= CALBC1_16MHZ;    //select 16MHz DCO
     BCSCTL1 |= DIVA_3;          //select divider for ACLK, VLO divide by 8 for ACLK;
     BCSCTL1 &= ~XTS;            //select low-frequency mode for LFXT1

     BCSCTL2 = 0x0;             //select DCOCLK for MCLK; MCLK = DCOCLK; SMCLK = DCOCLK;
divider for SMCLK = 1; use internal resistor;

     BCSCTL3 |= LFXT1S_2;//To use VLOCLK as low frequency clock, 12kHz
}
```

A 1-Hz timer is initialized with the following function:

```
void TimerAInitialise()
{
   TACTL = TASSEL0 + ID1 + TACLR;    //diviver = 4, clock source ACLK from VLOCLK, 12kHz
   TACCR0 = 375;                     //divide 12000Hz to 1Hz by 12000/4/8 * 375, divider
8 comes from the ratio from VLOCLK to ACLK
}
```

The timer should start after all modules have initialized and global interrupt is enabled with the following function:

```
void TimerAStart()
{
   TACTL |= TACLR;                                   //clear the TAR

   while(TACTL & TACLR)
   {}

   TACTL |= MC0;                                     //the timer is started

   TACCTL0 |= CCIE;                                  //Module Interrupt enable
}
```

I^{2}C is initialized for communication with the bq78350, the clock rate for I^{2}C is configured to 80 kHz.

```
void I2CInitialise()
{
   P1SEL |= BIT6 + BIT7;           // Assign I2C pins to USCI_B0, P1.6 for SCL and P1.7 for SDA
   P1SEL2|= BIT6 + BIT7;           // Assign I2C pins to USCI_B0, P1.6 for SCL and P1.7 for SDA

   ADC10AE0 &= 0x3F;
   CAPD &= 0x3F;

   UCB0CTL1 |= UCSWRST;                 // Enable SW reset, hold USCI logic in reset state
   UCB0CTL0 = UCMODE_3 + UCSYNC;        //set to I2C mode, sync=1
   UCB0BR0 = 200;                       //80kHz clock rate
   UCB0BR1 = 0;

   UCB0I2CIE = 0;
   IE2 &= ~(UCB0TXIE + UCB0RXIE);               //disable interrupts

   UCB0CTL1 |= UCSSEL_2;
   UCB0CTL1 &= ~UCSWRST;
}
```

The UART is initialized idle line mode, no parity, 8 bit data, 1 stop bit. The baud rate is initialized to 128 kBd.

```
void UARTInitialise()
{
   P1SEL |= BIT1 + BIT2;
   P1SEL2 |= BIT1 + BIT2;
   ADC10AE0 &= 0xF9;
   CAPD &= 0xF9;            //set P1.1 to RXD, P1.2 to TXD

   UCA0CTL1 |= 1;
   UCA0CTL0 |= UCMODE_1;    //Select idle-line mode, Parity disable, 8-bit data, one stop
bit, asynchronous mode
   UCA0CTL1 |= UCSSEL_2 + UCRXEIE + UCBRKIE + UCDORM + UCSWRST;      //SMClock, UCRXEIE = 1,
UCBRKEIE = 1; UCDORM = 1; UCTXADDR = 0, UCTXBRK = 0, UCSWRST = 1;

   UCA0BR0 = 125;
   UCA0BR1 = 0;
   UCA0MCTL = 0;           //UCOS16 = 0; Baud Rate = 128000

   UCA0STAT = 0x0;

   UCA0CTL1 &= ~UCSWRST;
}
```

2 pins of port 2, P2.3 and P2.4 are initialized for shutdown detect and CAN bus activity detector, the port2 initialize code follows:

```
void Port2Init()
{
   //set P2.3 to shutdown control function
   P2SEL &= ~BIT3;
   P2SEL2 &= ~BIT3;                        //select the basic IO function
   P2DIR &= ~BIT3;                        //select the input function
   P2IES |= BIT3;                        //set the IFG on high to low transition
   P2IE |= BIT3;

   //set P2.4 to CAN Communication detector
   P2SEL &= ~BIT4;
   P2SEL2 &= ~BIT4;                        //select the basic IO function
   P2DIR &= ~BIT4;                        //select the input function
   P2IES |= BIT4;                        //set the IFG on high to low transition
   P2IE |= BIT4;
}
```

5.2 Main Loop

After the modules are initialized, the MCU enters into the main loop:

```
     while(1)
     {
        WDT_CLEAR;                                     //Clear the watchdog

        Delay10mS();

        CheckUARTTimeOut();                           //Check if an incomplete frame has received

        if (COMMAND_PROCESSING == CommandStatus)       //Check if there is any command need to be processed
        {
                if(!(Semaphore & SLEEPTIMER_SEMAPHORE))
                {
                        Semaphore |= SLEEPTIMER_SEMAPHORE;
                        SleepTimer = 0;                //Clear the SleepTimer, system goes to sleep mode once this
time is greater than 20
                        Semaphore &= ~SLEEPTIMER_SEMAPHORE;
                }
                ComposeCommand();
                ProcessCommand();                      //this two functions interpreted and handles the command
from the host
        }

                if (POWER_MODE_SLEEP == PowerMode)             //PowerMode is a global variable, it is set to
POWER_MODE_SLEEP in the timer interrupt routine which also counts the SleepTimer, once SleepTimer is counted up to
20 or greater than 20, the PowerMode is set to POWER_MODE_SLEEP
                {
                        I2CSendBytes(BQ78350, (unsigned char*)SleepCommand, 3, &SentByte);
           //bq78350 enters into sleep mode
                        DISABLE_INT;
                        DisableUART ();        //Disable UART
                        //P2OUT |= BIT2;        disable DCDC output

                        P2IE |= BIT4;          //enable interrupt for P2.4, this pin is used to detect CAN
BUS transaction, but the related interrupt should only be enabled when system goes into the sleep mode
                        while(P2IFG & BIT4)
                        {
                                P2IFG &= ~BIT4;
                        }
                //just in case of successive pulse causes entering the P2_ISR interrupt routine here, once
this happens,
                //the P2.4 interrupt will be disabled under LPM3 mode in the interrupt routine,

                //this causes the device never returns to active mode
                        ENTER_LPM3;//enter LPM3, meanwhile, the interrupt is enabled.
                }

                if (POWER_MODE_SLEEP_ACTIVE == PowerMode)     //this if statement handles the case when the
system is to revive to normal mode from sleep mode after a transaction on CAN bus is detected, the PowerMode is
only set to POWER_MODE_SLEEP_ACTIVE in the interrupt routine triggered by CAN bus transaction.
                {
                        Delay1S();
                        FrameResponse(FRAME_RESPONSE_AWAKENED);
                        PowerMode = POWER_MODE_ACTIVE;
                }
    }
```

5.3 Interrupt Handling

There are 3 interrupt routines implemented in the firmware. They are *TimerA interrupt*, *UART interrupt* and *P2.3, P2.4 interrupt*.

TimerA interrupt implements a *TickCounter*, this counter always increments by 1 in this routine, this timer is based on TimerA, which raises an interrupt every 1 second. *SleepDelay* is a preset constant for the MCU to check if another timer (*SleepTimer*) is expired or not, once the *SleepTimer* is greater than or equal to *SleepDelay*, the global variable *PowerMode* is set to *POWER_MODE_SLEEP*, which indicates the system has not received any command from the host for a predefined period and should go to *Sleep Mode* for power-saving purposes. The *SleepTimer* is cleared in the main loop once the global variable *PowerMode* is set to *POWER_MODE_SLEEP*.

```
#pragma vector = TIMER0_A0_VECTOR
__interrupt void Timer0_ISR(void)
{
     TickCounter++;

     if(!(Semaphore & SLEEPTIMER_SEMAPHORE) && (-1 != SleepDelay) && (POWER_MODE_ACTIVE ==
PowerMode))
     {
          Semaphore |= SLEEPTIMER_SEMAPHORE;
          SleepTimer++;
          Semaphore &= ~SLEEPTIMER_SEMAPHORE;

          if(POWER_MODE_ACTIVE == PowerMode)               //Entering the Active Mode
          {
               if( SleepDelay <= SleepTimer)
               {
                    SleepTimer = 0;
                    PowerMode = POWER_MODE_SLEEP;
               }
          }
     }
}
```

The *Port2 interrupt* is for P2.3 and P2.4 transition detection, P2.3 is for shutdown button detection, P2.4 is for CAN bus activity detection.

```
#pragma vector = PORT2_VECTOR
__interrupt void Port2_ISR(void)
{
     unsigned int SentByte, I;

     if(P2IFG & BIT3)       //Check whether shutdown button has been pressed
     {
          while(BIT3 & P2IFG)          //this is for clearance successive transition
caused by chattering
          {
               P2IFG &= ~BIT3;
          }
          I2CInitialise();
          I2CSendBytes(BQ78350, (unsigned char*)ShutdownCommand, 3, &SentByte);
                                                   //send command to shutdown bq78350
          Delay10mS();
          Delay10mS();
          I2CSendBytes(BQ78350, (unsigned char*)ShutdownCommand, 3, &SentByte);
                                                   //send command twice to shutdown bq78350
in case the device is sealed
          while(1)          //the dead loop here is never executed as the power of MCU
has lost when bq78350 executes the shutdown command.
          {
          }
     }

     if(BIT4 & P2IFG)     //This if statement is for detecting the transaction on CAN bus
     {
          P2IE &= ~BIT4;                              //disable interrupt for P2.4

          PowerMode = POWER_MODE_SLEEP_ACTIVE;          //Define a transition stage from
SLEEP to ACTIVE
          SleepTimer = 0;
     }
          RESTORE_WATCHDOG;
          LPM3_EXIT;
}
```

The UART interrupt handling consists of two routines for receiving and transmitting, respectively. The transmission routine sends the byte in a global array *SendBuffer*. A global variable *SendBufferIndex* is used as a pointer to the byte in the array to be sent. The global variable *SendByteNumber* saves the number of remaining unsent bytes in the *SendBuffer* array. It is set before the transmit interrupt has been enabled in a function of void UARTSendBytes(unsigned char *Buffer, unsigned char NumberOfBytes).

```
#pragma vector = USCIAB0TX_VECTOR
__interrupt void UCA0_TX_ISR(void)
{
     while(!(UCA0TXIFG & IFG2))
     {}
     if(SendByteNumber)
     {
         UCA0TXBUF = SendBuffer[SendBufferIndex];
          SendBufferIndex++;
          SendByteNumber--;
     }else
     {
     SendBufferIndex = 0;
     SendBuffer = NULL;
     DISABLE_UART_SEND_INT;
     }
}
```

The routine for receiving is listed as follows: Local Variable *UCA0Status* stores the status register of UCA0, which is the engine for UART transaction, the variable *ReceivedByte* is used to save the byte in the UART buffer register. This routine first checks if it is in the expected condition by checking the global variable *BufferFull* and *CommandStatus*, as this routine is only allowed to enter when the *CommandStatus* indicates the system is in Idle or is in the process of receiving data, then checks if the UART engine is configured in idle line mode by checking the bits of UCMODE0 and UCMODE1 in UCA0CTL0 register, then reads the UCA0RXBUF and stores the byte to *ReceivedByte*, the first byte supposed to be the address defined in the UART engine. If it does not match with the predefined address, it quits the routine or otherwise exits the low power mode if it is in this mode before receiving this byte, and then prepares to receive the following bytes by initializing the related variables like *CommandStatus*, *DataCounterInFrame*, *ParseBufferIndex* and *UARTCounter*. The second byte is used to indicate the length of the frame, it is utilized by the receive routine to determine if all bytes in this transaction have been received, any byte more than the indicated frame length is ignored, and if enough bytes have not been received within a predefined time period, the function *CheckUARTTimeOut* in the main loop will terminate the receiving process and send *UART Timeout Frame* to the host.

```
#pragma vector = USCIAB0RX_VECTOR
__interrupt void UCA0_RX_ISR(void)
{
        DISABLE_INT;
        static unsigned char UCA0Status = 0;
        static unsigned char ReceivedByte = 0;

        if (1 == BufferFull && COMMAND_PROCESSING == CommandStatus)
        {
                goto IntRet;
        }

        if(UCMODE_1 == (UCA0CTL0 & (UCMODE0 | UCMODE1)))        //idle line mode identification
        {

                UCA0Status = UCA0STAT;
                //Any Receive Error, return

                if(UCRXERR & UCA0Status)
                {
                        ReceivedByte = UCA0RXBUF;
                        goto IntRet;
                }

                ReceivedByte = UCA0RXBUF;

                if (UCIDLE & UCA0Status)
                //Address byte received
                {

                        if(LOCAL_ADDRESS != ReceivedByte)
                        {
                                UCA0CTL1 |= UCDORM;
                                goto IntRet;
                        }
```

```
                                    if(UCDORM & ~UCA0CTL1)
                            //if UCDORM is already cleared, return;
                                    {
                                            ReceivedByte = UCA0RXBUF;
                                            goto IntRet;
                                    }

                                    if(POWER_MODE_SLEEP == PowerMode)
                    //if address is correct, then exit low power mode.
                                    {
                                            RESTORE_WATCHDOG;
                                            LPM3_EXIT;
                                            SleepTimer = 0;
                                            //enforce SleepTimer to be 0, no need to acquire semaphore
                                            PowerMode = POWER_MODE_ACTIVE;
                                    }
                                    UCA0CTL1 &= ~UCDORM;
                                    //Local Address Identified, prepare to receive following bytes.
                                    CommandStatus = COMMAND_RECEIVING;
                                    DataCounterInFrame = 0;
                                    ParseBufferIndex = ReceiveBufferIndex;
                                    UARTCounter = TickCounter;
                            //Keep UARTCounter synchronized to TickCounter;
                            }else
                            {
                                    if (ReceiveBufferIndex < BUFFER_SIZE)
                                    {
                                            ReceiveBuffer[ReceiveBufferIndex] = ReceivedByte;

                                            if (0 == DataCounterInFrame)
                                            {
                                                    FrameLength = ReceivedByte;
                                            }

                                            ReceiveBufferIndex++;
                            //ReceiveBufferIndex should be decremented outside this interrupt routine.
                                            DataCounterInFrame++;
                                            UARTCounter = TickCounter;
                    //Keep UARTCounter synchronized to TickCounter;

                                    }else
                                    {
                                            BufferFull = 1;
                                    //Clear BufferFull should also be done outside this routine.
                                    }
                            }

                            if (FrameLength + 2 == DataCounterInFrame && FrameLength != 0)    //check if a
            complete frame has been received.
                                    //The Frame for write command is like |Device
            Address|FrameLength|OperationType|Command1|Command2|...|Commandn|CheckSum|, FrameLength is the
            byte number from Operation to Commandn, start from 1
                                    //The Frame for receiving data is like |Device
            Address|FrameLength|OperationType|Command1|Command2|...|Commandn|ReadBytesCount|Checksum|,
            FrameLength includes one more byte for ReadByteCount.
                            {
                                    FrameLength = 0;
                                    DataCounterInFrame = 0;
                                    UCA0CTL1 |= UCDORM;
                                    CommandStatus = COMMAND_PROCESSING;
                                    UARTCounter = 0;
                    //Clear UARTCounter after one frame has been received.
                            }
                    }

            IntRet:
                    UCA0Status = UCA0STAT;
                    ENABLE_INT;
            }
```

```
             //The Frame for write command is like |Device
             Address|FrameLength|OperationType|Command1|Command2|...|Commandn|CheckSum|, FrameLength is the
             byte number from Operation to Commandn, start from 1
                                     //The Frame for receiving data is like |Device
             Address|FrameLength|OperationType|Command1|Command2|...|Commandn|ReadBytesCount|Checksum|,
             FrameLength includes one more byte for ReadByteCount.
                             {
                                     FrameLength = 0;
                                     DataCounterInFrame = 0;
                                     UCA0CTL1 |= UCDORM;
                                     CommandStatus = COMMAND_PROCESSING;
                                     UARTCounter = 0;
                     //Clear UARTCounter after one frame has been received.
                             }
                     }

             IntRet:
                     UCA0Status = UCA0STAT;
                     ENABLE_INT;
             }
```

6 Conclusion

The protocols discussed in this application report were implemented in the lab and run perfectly on the hardware platform in a real application. A DC/DC converter is still needed to provide the power to the ISO1050, and this converter should be disabled when the system transits to *Sleep Mode* to save energy. The LM25018 can be used for this DC/DC converter implementation.