Bateria virtual controlada pela Launchpad MSP430G2553

Filipe de Souza Freitas
FGA
Universidade de Brasília
Gama, Brasil

Rodrigo Magalhães Caires
FGA
Universidade de Brasília
Gama, Brasil
rodrigo_macrt@hotmail.com

Palavras-chave —Microcontrolador, MSP430, Percussão, Bluetooth, I2C, UART.

I. RESUMO

Sistema de controle utilizando de uma a duas pulseiras com acelerômetro para captar o movimento dos braços do usuário e emular o som correspondente a uma peça de um kit de percussão em uma plataforma de áudio (DAW).

II. INTRODUÇÃO

A prática musical é uma atividade bastante popular que desperta o interesse de grupos demográficos bastante diversificados e que se estende pelos mais variados tipos de instrumento e estilos musicais. Existem inúmeras inconveniências e dificuldades impostas pelo cotidiano que desincentivam o aprendizado e a prática de atividades musicais. Alguns fatores importantes a se considerar quanto a viabilidade deste hobby são: disponibilidade de tempo, espaço para treino e custos gerais relacionados. A proposta deste projeto é fornecer uma alternativa prática para o treino de instrumentos de percussão fazendo o uso de duas SmartBands com sensores de movimento.

III. DESENVOLVIMENTO

A partir do refinamento do código da comunicação UART, foi possível separar o mesmo em diversas funções para conveniência durante as manutenções. A intenção é de adicionar mais opções velocidades de clock para as próximas versões para testar e determinar qual taxa de transmissão de dados é a mais apropriada para evitar delay na resposta do aúdio e de adaptar a função de transmissão para os dados recebidos do sensor.

Como os problemas de coleta de dados durante a comunicação entre a Launchpad e a MPU6050 não foram resolvidos, não houve como testar o envio de dados recebidos. Porém, realizou-se um teste mais próximo do esperado para o projeto final utilizando um vetor de caracteres e um contador para simular a mudança dos dados transmitidos durante a execução. Ao final de cada ciclo, o bluetooth envia a string “COMPLETED ” para o dispositivo de destino e pode ser programado para repetir automaticamente ao início ou aguardar o clique no botão de reset, demonstrando o funcionamento tanto do envio contínuo de dados quanto da interrupção.

Para as etapas finais do projeto, será incluída uma versão mais sofisticada da função de temporização utilizando o Timer A que auxiliará na conversão dos dados de aceleração para posição relativa e será modificado o tipo do vetor de dados de char para signed short int para conciliar com os 16 bits de dados transmitidos pelo sensor para cada um dos três eixos disponíveis e levando em conta a direção do movimento relativo ao ponto de referência.

IV. DESCRIÇÃO DE HARDWARE

Para a execução do projeto, serão utilizados uma Launchpad MSP430G2553, um multisensor MPU6050 e um dispositivo bluetooth HC-05.

Serão utilizados dois protocolos de comunicação, um para cada periférico, sendo que cada um é realizado por um par diferente de pinos. Os protocolos de cada componente são definidos pelos fabricantes e as especificações da comunicação são definidas nos datasheets dos mesmos. A alimentação é paralela para ambos os módulos.

A comunicação I2C se caracteriza pela utilização de apenas dois pinos, um para o sincronismo do sistema(clock) e outro para transmissão e recepção de dados. Os pinos no MSP que contém acesso ao hardware responsável pela comunicação I2C são: P1.6 e P1.7, SCL e SDA respectivamente.
O pino SCL(P1.6) é responsável por estabelecer o sincronismo do sistema e o pino SDA(P1.7) pela transmissão de dados.

Para o HC-05, será utilizada a comunicação UART, realizada pelos pinos P1.1 (RX) e P1.2 (TX). O pino P1.1 tem a função de receber dados dos periféricos e, portanto, não será utilizado neste projeto já que o fluxo de dados será unidirecional no sentido da Launchpad para o bluetooth. O pino P1.2 é responsável pela transmissão de dados do mestre para os periféricos e será o único pino de comunicação ativo conectado ao bluetooth.

V. DESCRIÇÃO DE SOFTWARE

Comunicação I2C

Para realizar a comunicação I2C, através de informações do fabricante(Invesense), o protocolo é implementado através dos seguintes passos:

- Envio da condição de Start pelo master da comunicação, seguido do endereço e se a operação é de escrita ou leitura;
- O slave envia um sinal ACK.
- O master ao perceber o sinal ACK envia uma condição de stop, mais o endereço e o tipo de de operação caso for leitura.
- O slave envia o sinal de ACK e escreve os dados pedidos pelo master de determinado registrador.
- O master encerra a comunicação enviando os sinais NACK e STOP caso a comunicação seja para um byte, ou envia um sinal ACK e espera a transmissão dos dados para encerrar a transmissão.

Os registradores de controle para a comunicação I2C são:

1. UCB0CTL0 - Controla o modo de operação.
2. UCB0CTL1 - Controla parte do fluxo de operação como condições de START e STOP.
3. UCB0BR0 - LSB do registrador que define o clock da comunicação.
4. UCB0BR1 - MSB do registrador que define o clock da comunicação.
5. UCB0I2CIE - Enable das interrupções associadas à comunicação I2C.
6. UCB0STAT - Status da comunicação.
7. UCB0RXBUF - Buffer para receber os dados da comunicação.
8. UCB0TXBUF - Buffer para enviar os dados da comunicação.
9. UCB0I2COA - Endereço para o msp.
10. UCB0I2CSA - Endereço para slave da comunicação.

Para começar "setamos" os principais registradores responsáveis por ajustar o hardware através dos registradores: UCB0CTL0, UCB0CTL1,UCB0BR0 ,UCB0BR1, UCB0I2CIE UCB0I2CSA, da seguinte forma:

- P1SEL |= BIT6 + BIT7; - "Conectar" os pinos no hardware da comunicação I2C.
- P1SEL2|= BIT6 + BIT7; - "Conectar" os pinos no hardware da comunicação I2C.
- UCB0CTL1 |= UCSWRST; - "Resetar" o registrador de controle 1.
- UCB0CTL0 = UCMST + UCMODE_3 + UCSYNC; -Configurar o sincronismo entre os dois sistemas.
- UCB0CTL1 = UCSSEL_2 + UCSWRST; - Configurar para uso de referencia o clock master de subsistemas.
- UCB0BR0 = 10; - Configura o clock como sendo aproximadamente 100k Hz.
- UCB0BR1 = 0;
- UCB0CTL1 &= ~UCSWRST; -Reseta o bit de reset do controlador 1.

Após a definição do modo de operação, é necessário começar a comunicação através do sinal START, isto é feito "setando" com o auxílio da flag UCTXSTT mais a flag UCTR invertida(~UCTR) para leitura e não invertida para transmissão.
Quando for fazer leitura é necessário atribuir o valor no buffer UCB0RXBUF para uma variável isso limpa a flag de interrupção no registrador UCB0STAT. Para enviar é necessário escrever no registrador UCB0TXBUF.

Comunicação UART

Como não há o compartilhamento de um sinal de clock, ambos os dispositivos devem estar em acordo quanto à frequência de troca de informações. Para isso, tanto o dispositivo mestre quanto o escravo devem ser configurados sob a mesma baud rate (taxa de variação de sinais por segundo). Nesse caso, o módulo HC-05 já vem com a baud rate pré-definida de 9600 bits/s então basta configurar a Launchpad com a mesma taxa. Utilizando o datasheet da MSP430G2553 e definindo o clock da placa em 1MHz, podemos ver que o valor tabelado para uma

baud rate de 9600 nessas circunstâncias é configurado quando o registrador UCBR0 recebe 104.

Como os pinos da MSP430G2553 são multiplexados, eles podem exercer funções diferentes dependendo da sua configuração. No caso do protocolo UART, os pinos Rx e Tx se encontram nos pinos P1.1 e P1.2 respectivamente. Porém, por default, estes pinos são inicialmente definidos como GPIO, ou seja, como pinos de entrada ou saída de propósito geral. Para trocar as funcionalidades destes e outros pinos, utilizamos os registradores de 8 bits P1SEL e P1SEL2. Ao setar os bits 1 e 2 (referentes aos pinos P1.1 e P1.2, respectivamente) em ambos os registradores, habilitamos as funcionalidades RXD e TXD para a UART.

Em seguida, "setam" se os interrupt enables do receptor, do transmissor e o global. A flag do transmissor é setada automaticamente quando um dado é enviado, ou seja, quando o registrador UCA0TXBUF está vazio, e é "resetada" automaticamente quando um novo dado é escrito no mesmo registrador. Já a flag do receptor é setada automaticamente quando um dado é recebido e armazenado no registrador UCA0RXBUF e precisa ser "resetada" manualmente sempre que for receber um novo dado.

A transmissão de dados é executada ao se armazenar uma informação qualquer de 1 byte no registrador UCA0TXBUF, considerando que o mesmo esteja vazio a princípio e a que a flag de transmissão esteja resetada. O bit UCLISTEN do registrador UCA0STAT habilita um "curto" interno entre os pinos TX e RX, permitindo que qualquer dado enviado pelo pino P1.2 seja recebido pelo pino P1.1 e tem sua utilidade resumida para casos de testes e simulações para verificar se ambas funções estão funcionando corretamente.

VI. Resultados

A comunicação I2C implementada não retornou dados do acelerômetro, em uma tentativa utilizando o LED vermelho da launchpad verificou-se que o msp430 não recebe dados do sensor embora as funções de início, e condições de stop e start são ativadas pois o led foi colocado para ascender quando passasse pelas funções de "checagem" das flags de condições START, STOP e NACK presentes no registrador STAT.

Para testar o envio contínuo de dados da MSP para o dispositivo destino, outro computador neste caso, utilizou-se um vetor de caracteres, um contador para o vetor e outro contador para fazer um delay primitivo. O vetor foi inicializado com a string de 10 caracteres "COMPLETED " e a função Transmitir foi colocado dentro de um loop infinito. Após o envio de cada caractere, realiza-se uma contagem de 30 mil ciclos e incrementa-se o contador "j" para mover para o próximo caractere do array. Ao terminar a leitura da string, reseta-se a variável j para recomeçar a leitura da vetor. O contador de 30 mil ciclos foi utilizado apenas para facilitar a leitura dos dados no terminal durante o teste e não deve aparecer na versão final do projeto. Em seguida, executou-se um segundo teste para interromper a transmissão assim que a string fosse completamente enviada, podendo ser chamada repetidas vezes ao clicar no botão reset.

VII. Conclusão

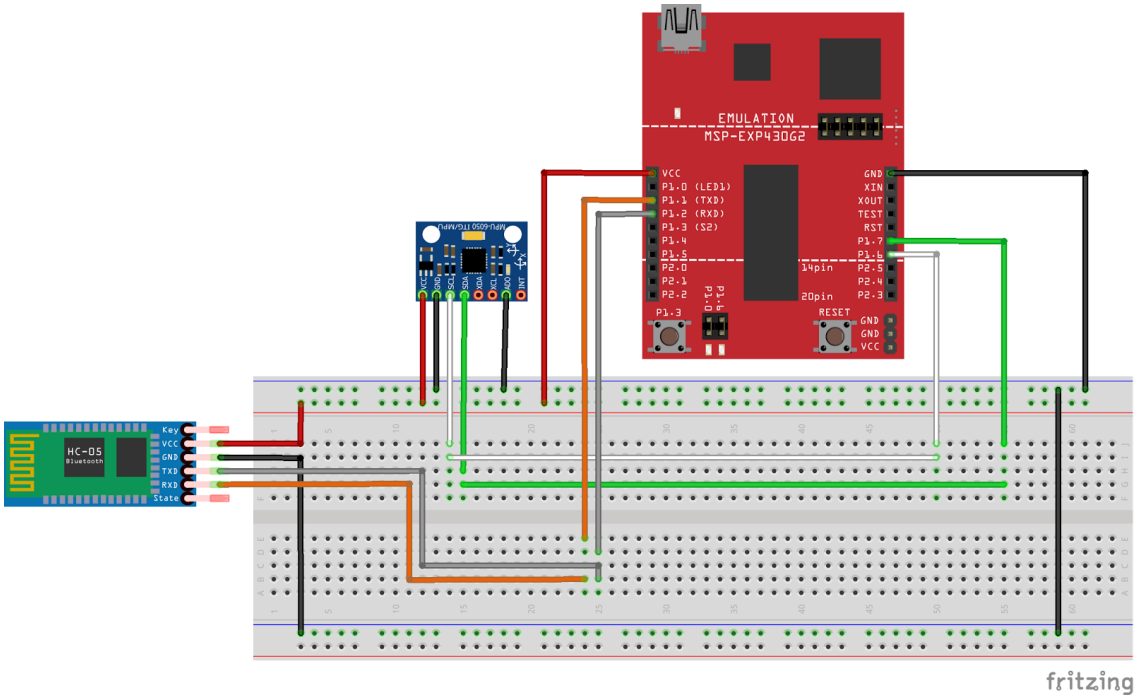
Conclui-se que a comunicação UART foi realizada com sucesso, o envio de dados é possível com o msp430g2553 através do padrão bluetooth com o auxílio do dispositivo HC-05, tornando possível a etapa do projeto onde enviamos palavras que determinam uma ação a ser executada pelo dispositivo externo ao sistema msp,acelerômetro e bluetooth.

VIII. Referências

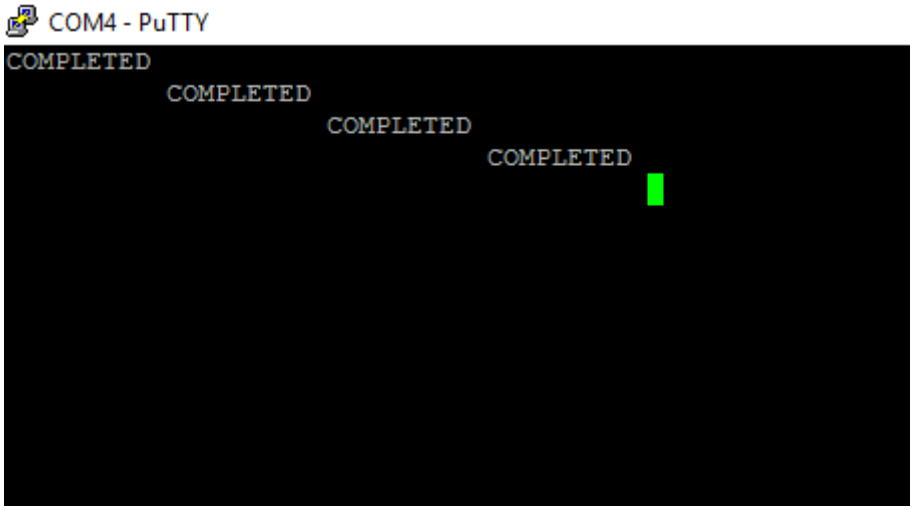
[1] Texas Instruments, ChronosDrums. Disponível em: http://processors.wiki.ti.com/index.php/ChronosDrums
[2] Xanthium. Serial Communication Using MSP430 UART(USCI-A). http://xanthium.in/Serial-Communication-MSP430-UART-USCI_A
[3] SLAU144 USER GUIDE.
[4] MPU6050 DATASHEET.
[5] HC-05 DATASHEET.

ANEXOS

ANEXO I



ANEXO II



ANEXO III

```
#include "msp430g2553.h"

      int tempo = 0;

      int j = 0;

      char word[10] = {'C','O','M','P','L','E','T','E','D',' '};
```

```
void Transmitir(int tempo, char sinal){

 UCA0TXBUF = sinal;                  // Transmit a byte
 while(tempo < 30000){
    tempo++;
 }

}

void setClock(void){

if (CALBC1_1MHZ==0xFF)   // If calibration constant erased
     {
        while(1);          // do not load, trap CPU!!
     }

   DCOCTL  = 0;              // Select lowest DCOx and MODx settings
   BCSCTL1 = CALBC1_1MHZ;    // Set range
   DCOCTL  = CALDCO_1MHZ;    // Set DCO step + modulation

}

void setUART(void){
```

```
 P1SEL  |=  BIT1 + BIT2;  // P1.1 UCA0RXD input
 P1SEL2 |=  BIT1 + BIT2;  // P1.2 UCA0TXD output
 UCA0CTL1 |=  UCSSEL_2 + UCSWRST;  // USCI Clock = SMCLK,USCI_A0 disabled
 UCA0BR0   =  104;                 // 104 From datasheet table-
 UCA0BR1   =  0;                   // -selects baudrate =9600,clk = SMCLK
 UCA0MCTL  =  UCBRS_1;             // Modulation value = 1 from datasheet
 //UCA0STAT |=  UCLISTEN;            // loop back mode enabled
 UCA0CTL1 &= ~UCSWRST;             // Clear UCSWRST to enable USCI_A0

}

void setInterrupt(void){

 IE2 |= UCA0TXIE;                  // Enable the Transmit interrupt
 IE2 |= UCA0RXIE;                  // Enable the Receive  interrupt
 _BIS_SR(GIE);                     // Enable the global interrupt

}

void main(void)
{

WDTCTL = WDTPW + WDTHOLD; // Stop the Watch dog
```

```
setClock();
setUART();
setInterrupt();

//---------------- Configuring the LED's ----------------------//

 P1DIR  |=  BIT0 + BIT6;  // P1.0 and P1.6 output
 P1OUT  &= ~BIT0 + BIT6;  // P1.0 and P1.6 = 0

 //UCA0TXBUF = 'X';                  // Transmit a byte
 while(1){



    Transmitir(0, word[j]); //RECEPTOR PUTTY COM4
    j++;

    if(j == 11){
      break;
    }

 }

 _BIS_SR(LPM0_bits + GIE);         // Going to LPM0
}
```

```
#pragma vector = USCIAB0TX_VECTOR

__interrupt void TransmitInterrupt(void)

{

  P1OUT  ^= BIT0;//light up P1.0 Led on Tx

  IFG2 &= ~UCA0TXIFG;

}



#pragma vector = USCIAB0RX_VECTOR

__interrupt void ReceiveInterrupt(void)

{

  P1OUT  ^= BIT6;     // light up P1.6 LED on RX

  IFG2 &= ~UCA0RXIFG; // Clear RX flag

}
```

Anexo IV

```
/*
* I2C_Accel_MPU6050
*
* Version 1:  Read raw accelerometer X, Y and Z data continuously
*
*  P1.6              UCB0SCL
*  P1.7              UCB0SDA
*
* MPU-6050 Accelerometer & Gyro
* NOTE:  Default state of the MPU-6050 is SLEEP mode.
*       Wake up by writing 0x00 to the PWR_MGMT_1 register
* NOTE:  No-name version from Amazon has a 5V to 3.3V regulator, and Vcc MUST be 5V !!!
*       10-kOhm pull-up resistors are included on the board
*       330-Ohm resistors are included in series on SCL and SDA
*             (safe to connect P1.6 & P1.7 directly to SCL and SDA)
*
* Slave address:   0x68 (AD0=0) or 0x69 (AD0=1)
*
* Z-data buffer addresses:
*              0x3B ACCEL_XOUT_H R ACCEL_XOUT[15:8]
*              0x3C ACCEL_XOUT_L R ACCEL_XOUT[ 7:0]
*              0x3D ACCEL_YOUT_H R ACCEL_YOUT[15:8]
*              0x3E ACCEL_YOUT_L R ACCEL_YOUT[ 7:0]
```

```
*               0x3F ACCEL_ZOUT_H R ACCEL_ZOUT[15:8]
*               0x40 ACCEL_ZOUT_L R ACCEL_ZOUT[ 7:0]
*
* pins not used:   INT (interrupt for data ready in the 1024-byte FIFO bufer)
*               XCL, XDA (external clock and data lines for MPU-6050 I2C bus)
*
* Reading the raw values: disable sleep mode
*               0x6B PWR_MGMT_1 --> set to 0
*
*/
#include <msp430g2553.h>

unsigned char RX_Data[6];
unsigned char TX_Data[2];
unsigned char RX_ByteCtr;
unsigned char TX_ByteCtr;

int xAccel;
int yAccel;
int zAccel;

unsigned char slaveAddress = 0x68;  // Set slave address for MPU-6050
                // 0x68 for ADD pin=0
                // 0x69 for ADD pin=1

const unsigned char ACCEL_XOUT_H = 0x3B;    // MPU-6050 register address
const unsigned char ACCEL_XOUT_L = 0x3C;    // MPU-6050 register address
const unsigned char ACCEL_YOUT_H = 0x3D;    // MPU-6050 register address
const unsigned char ACCEL_YOUT_L = 0x3E;    // MPU-6050 register address
const unsigned char ACCEL_ZOUT_H = 0x3F;    // MPU-6050 register address
const unsigned char ACCEL_ZOUT_L = 0x40;    // MPU-6050 register address

void i2cInit(void);
void i2cWrite(unsigned char);
void i2cRead(unsigned char);
long map(long,long,long,long,long);


int main(void)
{

  WDTCTL = WDTPW + WDTHOLD;             // Stop WDT


  // Set clock speed (default = 1 MHz)
  BCSCTL1 = CALBC1_1MHZ;                // Basic Clock System CTL (1,8,12 16_MHZ
available)
  DCOCTL  = CALDCO_1MHZ;                // Digitally-Controlled Oscillator CTL

  // Initialize the I2C state machine
  i2cInit();

  // Wake up the MPU-6050
  slaveAddress = 0x68;                  // MPU-6050 address
  TX_Data[1] = 0x6B;                    // address of PWR_MGMT_1 register
  TX_Data[0] = 0x00;                    // set register to zero (wakes up the MPU-6050)
  TX_ByteCtr = 2;
  i2cWrite(slaveAddress);
```

```
  while (1)
  {
     // Point to the ACCEL_ZOUT_H register in the MPU-6050
     slaveAddress = 0x68;                  // MPU-6050 address
     TX_Data[0] = 0x3B;                 // register address
     TX_ByteCtr = 1;
     i2cWrite(slaveAddress);

     // Read the two bytes of data and store them in zAccel
     slaveAddress = 0x68;                  // MPU-6050 address
     RX_ByteCtr = 6;
     i2cRead(slaveAddress);
     xAccel  = RX_Data[5] << 8;            // MSB
     xAccel |= RX_Data[4];                 // LSB
     yAccel  = RX_Data[3] << 8;            // MSB
     yAccel |= RX_Data[2];                 // LSB

     zAccel  = RX_Data[1] << 8;            // MSB
     zAccel |= RX_Data[0];                 // LSB

     __delay_cycles(20000);
  }
}

void i2cInit(void)
{
  // set up I2C module
  // set up I2C pins
  P1SEL |= BIT6 + BIT7;                  // Assign I2C pins to USCI_B0
  P1SEL2|= BIT6 + BIT7;                  // Assign I2C pins to USCI_B0
  UCB0CTL1 |= UCSWRST;                   // Enable SW reset
  UCB0CTL0 = UCMST + UCMODE_3 + UCSYNC;          // I2C Master, synchronous mode
  UCB0CTL1 = UCSSEL_2 + UCSWRST;         // Use SMCLK, keep SW reset
  UCB0BR0 = 10;                  // fSCL = SMCLK/12 = ~100kHz
  UCB0BR1 = 0;
  UCB0CTL1 &= ~UCSWRST;                  // Clear SW reset, resume operation
}

void i2cWrite(unsigned char address)
{
  __disable_interrupt();
  UCB0I2CSA = address;                   // Load slave address
  IE2 |= UCB0TXIE;               // Enable TX interrupt
  while(UCB0CTL1 & UCTXSTP);         // Ensure stop condition sent
  UCB0CTL1 |= UCTR + UCTXSTT;         // TX mode and START condition
  __bis_SR_register(CPUOFF + GIE);       // sleep until UCB0TXIFG is set ...
}

void i2cRead(unsigned char address)
{
  __disable_interrupt();
  UCB0I2CSA = address;                   // Load slave address
  IE2 |= UCB0RXIE;               // Enable RX interrupt
  while(UCB0CTL1 & UCTXSTP);         // Ensure stop condition sent
  UCB0CTL1 &= ~UCTR;             // RX mode
  UCB0CTL1 |= UCTXSTT;                   // Start Condition
  __bis_SR_register(CPUOFF + GIE);       // sleep until UCB0RXIFG is set ...
}
```

```
#pragma vector = USCIAB0TX_VECTOR
__interrupt void USCIAB0TX_ISR(void)
{
  if(UCB0CTL1 & UCTR)               // TX mode (UCTR == 1)
  {
     if (TX_ByteCtr)                // TRUE if more bytes remain
     {
        TX_ByteCtr--;               // Decrement TX byte counter
        UCB0TXBUF = TX_Data[TX_ByteCtr];    // Load TX buffer
     }
     else                           // no more bytes to send
     {
        UCB0CTL1 |= UCTXSTP;            // I2C stop condition
        IFG2 &= ~UCB0TXIFG;         // Clear USCI_B0 TX int flag
        __bic_SR_register_on_exit(CPUOFF);  // Exit LPM0
     }
  }
  else // (UCTR == 0)               // RX mode
  {
     RX_ByteCtr--;                  // Decrement RX byte counter
     if (RX_ByteCtr)                // RxByteCtr != 0
     {
        RX_Data[RX_ByteCtr] = UCB0RXBUF;    // Get received byte
        if (RX_ByteCtr == 1)            // Only one byte left?
        UCB0CTL1 |= UCTXSTP;            // Generate I2C stop condition
     }
     else                           // RxByteCtr == 0
     {
        RX_Data[RX_ByteCtr] = UCB0RXBUF;    // Get final received byte
        __bic_SR_register_on_exit(CPUOFF);  // Exit LPM0
     }
  }
```