I2C Receive Application Note

Drew Currie

February 2024

1 Introduction

This application note describes the software slave implementation of I^2C serial communication on the MSP430FR2355 micro-controller. This application note also includes example code showing the implementation of I^2C master communication on the MSP430FR2355 micro-controllers with corresponding MSP430FR2310 in slave mode.

The example provides I^2C communication for the follow cases:

- Master transmit
- Slave receive
- Implementation of both with Interrupt Service Routines (ISR)

The main features of an I^2C bus are as follows:

- Two lines are required on the bus. One to carry the clock signal and one for the data signal.
- The ability for the master to interface with multiple slave devices on the same bus with addressing.
- Communication up to 400kbps.
- External 10k ohm pull-up resistors to a 3.3v(or 5v for TTL) rail.

2 Interface Overview

The Inter-Integrated Circuit I^2C serial communication protocol was originally developed at Philips Semiconductors. I^2C communication allows for multiple master devices and multiple slave devices. This requires two wires, Clock (SCL) and Data (SDA). In a complete I^2C interface there must be at least one master and one slave.

After the start condition, the master will send a byte with most significant bit (MSB) first on the Data Line (SDA) along with eight Clock Pulses(SCL). The first seven bits of the first byte are the seven bit address of the slave device. The slave will only respond to the master if the seven bit address matches the seven bit address of the slave. (*Note:* The MSP430 series of processors can have up to 4 hardware slave addresses at a time) The eighth bit of the first byte represents the Read or Write (R/W) bit of the transmission. Assuming the eighth bit is low the slave enters into read mode to receive data from the master. The slave will read this data into it's own internal memory. If the eighth bit is a high the master is requesting data to be read from the slave's internal memory. The master is the device creating the SCL pulses for both of these communication examples.

If the slave receives its own address, the slave should return a valid acknowledge (ACK) to the master to state that the slave has received data successfully (In this case the data is it's own address). An example timing diagram is shown below of the initial data transmission.

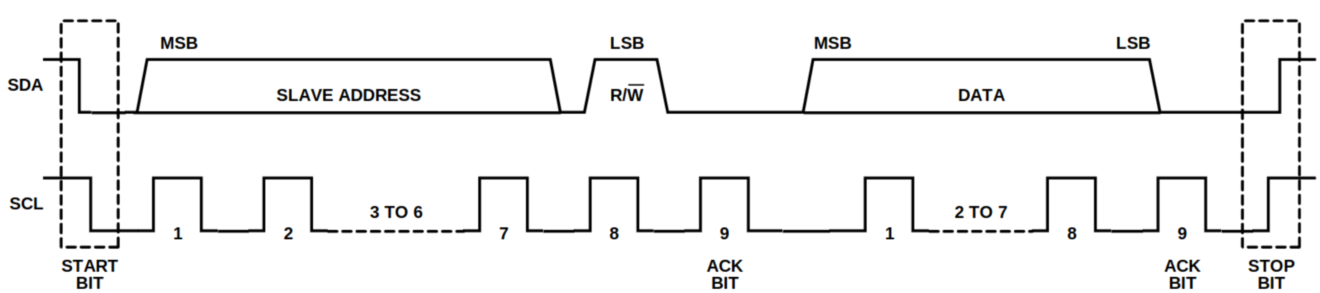

Figure 1: I^2C Timing Diagram

3 Acknowledge and No Acknowledge

If the slave address matches the address sent by the master, the slave must send an ACK back. Otherwise it should send a NACK or not respond. During the transfer of data on the I^{2}C bus the receiving device always generates the ACK or NACK. For example if a MSP430FR2355 is sending two bytes of data to a MSP430FR2310 configured as a slave device the MSP430FR2310 will generate the ACK or NACK after every byte.

If the master receives a NACK from a slave device the master should then generate a stop condition and abort the data transfer to prevent data corruption.

4 Transmitting Data

In the I^{2}C communication system an ISR or a polled loop implementation the microcontroller must decide whether to transmit or receive data. For the slave device, this depends on the R/W bit sent by the master. Based on this bit the slave must either receive the data or write data out on every clock pulse, and provide an acknowledge or listen for an acknowledge on the ninth clock pulse.

In the example used so far this means the MSP430FR2310 must provide the data when requested by the master and receive data from the master when provided data. This can be achieved with either a polling loop or an ISR. However, the use of an ISR is recommended.

5 Example Circuit

The circuit assembled to test the code provided in Appendix A, is shown below in Figure 2. In this example, the MSP430FR2355 is configured as the master with a MSP430FR2310 as the slave device. Connecting these two devices via an I^{2}C serial bus, data is transferred from the master to the slave.

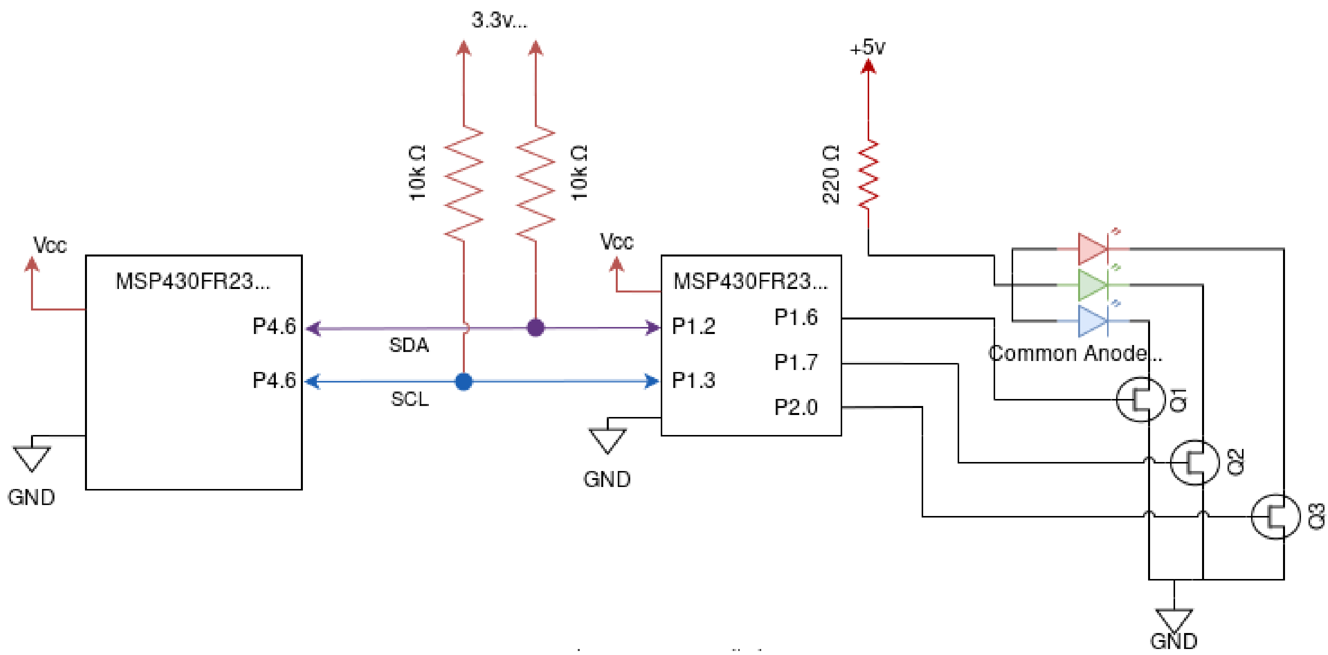

Figure 2: I^{2}C Example Circuit Diagram

In the code, the master will write out the hexadecimal ASCII equivalent of the color to be set on the RGB LED. An example logic analyzer screenshot is provided below where the master sent the packet to change the color to blue. When the slave receives this, the color of the LED is changed.

6 Slave Code

The MSP430FR2310 has been configured to operate in I^{2}C slave mode. This is achieved by configuring the Enhanced Universal Serial Communication Interface(eUSCI) peripheral on the MSP430FR2310 to be in I^{2}C slave mode. The full example code for the circuit is provided in Appendix A. This was derived from the code provided in the MSP430FR2xxx family data-sheet, in Chapter 24.

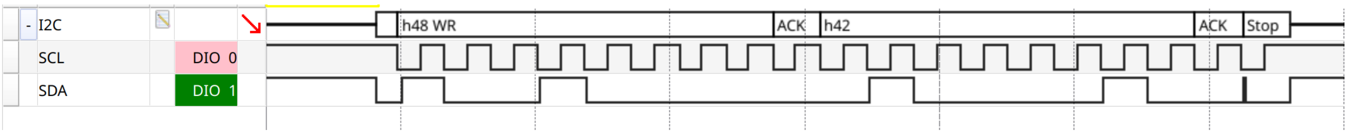

Figure 3: I^2C Example Waveform

```
// -- 1. Put eUSC_B0 into software reset
   UCB0CTLW0 |= UCSWRST;         // SW reset


// -- 2. Configure eUSCI_B0
/*
Put EUSCI0 into I2C mode, put into slave mode, set I2C address, and
enable own address.
*/
   UCB0CTLW0 |= UCMODE_3;
   UCB0CTLW0 &= ~UCMST;
   UCB0I2COA0 = 0x0048;
   UCB0I2COA0 |= UCOAEN;
   UCB0CTLW0 &= ~UCTR;
   UCB0CTLW1 &= ~UCASTP0;
   UCB0CTLW1 &= ~UCASTP1;

// -- 3. Configure ports
/*
Set P1.3 as I2C Clock, set P1.2 as I2C Data
*/
   P1SEL1 &= ~BIT3;
   P1SEL0 |= BIT3;

   P1SEL1 &= ~BIT2;
   P1SEL0 |= BIT2;
// -- 4. Take EUSCI_B0 out of software reset
   UCB0CTLW0 &= ~UCSWRST;
// -- 5. Enable local I2C receive interrupt and global interrupts
   UCB0IE |= UCRXIE0;
   __enable_interrupt();


```

In this example code the MSP430FR2310 has been configured with eUSCI (peripheral A0) in I^2C slave mode. This requires the eUSCI to be put into mode 3 (UCMODE_3)which determines the type of serial communication, clearing the master (UCMST) control bit, and setting the enable own address bit (UC0AEN).

Once these configuration bits have been set, the device is ready to operate in slave mode for I^2C slave receive mode. To respond with data, the system must change mode to transmit from receive. (An example of slave transmit is not provided in this App Note) This is achieved with the same configuration as the MSP430FR2355 for transmit.

7 Conclusion

The I^2C communication protocol is a robust communication system that quickly and easily allows devices to communicate. Implementing the I^2C communication on the MSP430FR2xxx series of microcontrollers allows for more complicated embedded systems to be developed.

A

Provided Slave and Master code for example circuit. MSP430FR2310 Slave Code:

```c
/*-------------------------------------------------------------------------------
 I2C Slave receive code
 Author: Drew Currie
 Date Created: 02/10/2024
 Last date editted: 02/17/204
 Purpose:
    Receive an ASCII value of either R, G, or B and change the output color of
    the RGB LED accordingly.
-------------------------------------------------------------------------------*/
#include "msp430fr2310.h"
#include <msp430.h>

volatile char DataIn = '0';
int main(void) {
  WDTCTL = WDTPW | WDTHOLD; // stop watchdog timer

  // -- 1. Put eUSC_B0 into software reset
  UCB0CTLW0 |= UCSWRST; // SW reset

  // -- 2. Configure eUSCI_B0
  /*
   Put EUSCI0 into I2C mode, put into slave mode, set I2C address, and
   enable own address.
   */
  UCB0CTLW0 |= UCMODE_3;
  UCB0CTLW0 &= ~UCMST;
  UCB0I2COA0 = 0x0048;
  UCB0I2COA0 |= UCOAEN;
  UCB0CTLW0 &= ~UCTR;
  UCB0CTLW1 &= ~UCASTP0;
  UCB0CTLW1 &= ~UCASTP1;

  // -- 3. Configure ports
  /*
   Set P1.3 as I2C Clock, set P1.2 as I2C Data
   */
  P1SEL1 &= ~BIT3;
  P1SEL0 |= BIT3;

  P1SEL1 &= ~BIT2;
  P1SEL0 |= BIT2;

  PM5CTL0 &= ~LOCKLPM5; // Turn on GPIO

  // -- 4. Take EUSCI_B0 out of software reset
  UCB0CTLW0 &= ~UCSWRST;

  // RGB LED pins
  // P2.0 -> Red, P1.7 -> Green, P1.6 -> Blue
  P1DIR |= BIT7;
  P1DIR |= BIT6;
  P2DIR |= BIT0;
  // -- 5. Enable local I2C receive interrupt and global interrupts
```

```c

  UCB0IE |= UCRXIE0;    // ENABLE I2C Rx0
  __enable_interrupt(); // enable maskable IRQs

  while (1) // Main loop
  {
    // Determine which ASCII character was received from the master
    switch (DataIn) {
    case 'R':
      P1OUT &= ~BIT7;
      P1OUT &= ~BIT6;
      P2OUT |= BIT0;
      break;
    case 'G':
      P2OUT &= ~BIT0;
      P1OUT &= ~BIT6;
      P1OUT |= BIT7;
      break;
    case 'B':
      P2OUT &= ~BIT0;
      P1OUT &= ~BIT7;
      P1OUT |= BIT6;
      break;
    default:
      break;
    }
    UCB0IE |= UCRXIE0; // Enable I2C Rx0
  }
  return 0;
}

/*
Begin I2C Interrupt Service Routine
    -Receive data from master and save in the DataIn variable.
*/

#pragma vector = EUSCI_B0_VECTOR // Triggers when RX buffer is ready for data,
                                 // after start and ack
__interrupt void EUSCI_B0_I2C_ISR(void) {
  DataIn = UCB0RXBUF; // Store data in variable
}
```

MSP430FR2355 Master Code:

```c


#include <msp430.h>
#include <stdint.h>

#define Slave_Address 0x048

const char I2C_Message[] = {'R', 'G', 'B'};
volatile unsigned int ColorIndex;
int main(void) {

  volatile uint32_t i;

```

```c
    // Stop watchdog timer
    WDTCTL = WDTPW | WDTHOLD;

    /*
    Configure I2C for master transmit mode
    */
    UCB1CTLW0 |= UCSWRST;  // Put UCB1CTLW0 into software reset
    UCB1CTLW0 |= UCSSEL_3; // Select mode 3
    UCB1BRW = 10;          // Something useful

    UCB1CTLW0 |= UCMODE_3; // Mode 3
    UCB1CTLW0 |= UCMST;    // Master
    UCB1CTLW0 |= UCTR;     // Transmit mode

    UCB1CTLW1 |= UCASTP_2; // Autostop enabled

    //----- P4.6 and P4.7 for I2C ---
    P4SEL1 &= ~BIT7;
    P4SEL0 |= BIT7;

    P4SEL1 &= ~BIT6;
    P4SEL0 |= BIT6;
    //--------------------------
    PM5CTL0 &= ~LOCKLPM5; // Take out of low power mode

    UCB1CTLW0 &= ~UCSWRST; // Take out of Software Reset

    UCB1IE |= UCTXIE0; // Enable TX interrupt
    UCB1IE |= UCRXIE0; // Enable RX interrupt

    __enable_interrupt();
    /*
    End I2C Setup
    */
    while (1) {

      /*Transmit Slave address and one byte of data*/
      UCB1TBCNT = 1;
      UCB1I2CSA = Slave_Address; // Set the slave address in the module
                                 //...equal to the slave address
      UCB1CTLW0 |= UCTR;    // Put into transmit mode
      UCB1CTLW0 |= UCTXSTT; // Generate the start condition
      for (i = 65000; i > 0; i--) {
        // Delay
      }
      for (i = 65000; i > 0; i--) {
        // Delay
      }
      for (i = 65000; i > 0; i--) {
        // Delay
      }
      for (i = 65000; i > 0; i--) {
        // Delay
      }
      for (i = 65000; i > 0; i--) {
        // Delay
      }
```

```
    for (i = 65000; i > 0; i--) {
      // Delay
    }
    ColorIndex++;
    if (ColorIndex > 2) {
      ColorIndex = 0;
    }
  }
}

#pragma vector = EUSCI_B1_VECTOR
__interrupt void EUSCI_B1_I2C_ISR(void) {
  UCB1TXBUF = I2C_Message[ColorIndex]; // Send the next byte in the
                                       // I2C_Message_Global string
}

```