Laboratory Worksheet #09
I2C Serial Communication Exercise

This worksheet provides some understanding of the SFRs and code associated with serial communication between the microcontroller and the peripheral devices.

The SMB0CN special function register is bit addressable. The individual bits indicate the status of a serial communication process. For the SMB0CN bits indicated below, identify the associated memory location. Note that the bits are not listed in the order 0-7. Refer to the c8051F020.h header file in Appendix A.

```
 /*Individual bits of the SMB0CN 0xC0 register */
__sbit __at ________ BUSY; /*SMBUS0 BUSY*/
__sbit __at ________ ENSMB; /*SMBUS0 ENABLE*/
__sbit __at ________ STA; /*SMBUS0 START FLAG*/
__sbit __at ________ STO; /*SMBUS0 STOP FLAG*/
__sbit __at ________ SI; /*SMBUS0 SIERRUPT PENDING FLAG*/
__sbit __at ________ AA; /*SMBUS0 ASSERT/ACKNOWLEDGE FLAG*/
__sbit __at ________ SMBFTE; /*SMBUS0 FREE TIMER ENABLE*/
__sbit __at ________ SMBTOE; /*SMBUS0 TIMEOUT ENABLE*/
```

(Note: The lines above are to help you keep track of the sbits used with the I2C bus. They are already defined in C0851F020.h and shouldn't be part of you code.)

Pseudo-code which describes a write operation from a Master device to a Slave device (Microcontroller to sensor/LCD) and a read operation form the Slave device to the Microcontroller are on the next two pages. The c-functions that perform Read/Write operations are provided at the end of the document. These functions are also provided in the i2c.h header file, available on the home page of the LMS website and in the appendix of the manual. The pseudo-code represents functions that call the functions seen in the provided code. It is recommended that you look through the header file or attached when doing the worksheet. When reviewing these functions in the laboratories, note the use of the same prototypes (i.e. function name, parameter order, etc.) as seen in this worksheet.

These functions process the data in the buffer in the same order it is processed on the bus. For example, the first function writes `buffer[0]` first, then `buffer[1]`, up to `buffer[num_bytes{1]`. The second function saves (reads) the first byte read to `buffer[0]`, the second to `buffer[1]`, etc.

For the for-loops in these two functions, remember that you do not want to read/write `buffer[num_bytes { 1]` inside the loop since this is the last byte in the buffer. The final byte will be written/read in the `write_and_-stop()` or `read_and_stop()` call at the end of the functions.

Exercise 1: Serial write operation

In the following pseudo-code, interpret the statements and provide the equivalent c-code, using proper syntax. Assume the address of the device is 0xA8 and that you are writing 4 bytes of data, starting at register 0 and writing sequentially. On the lines provided, write the c-code. For any function call that requires an argument (variable), indicate the actual value of the variable (except for the data itself).

```
// Write data to the I2C bus
//
// Parameters:
// unsigned char addr { address of the device that will be written to
// unsigned char start_reg { first register that will be written
// unsigned char* buffer { array of data to be written
// unsigned char num_bytes { number of elements in the array
void i2c_write_data(unsigned char addr, unsigned char start_reg, unsigned char *buffer,
unsigned char num_bytes)
```

{

Start I2C transfer

```
i2c_start();
```

Write the device address and indicate a write operation on the bus

```
i2c_write(addr & ~0x01); //addr is 0xA8
```

Write the start register

```
i2c_write(start_reg); //start_reg is 0
```

*/*** You will need to use a loop for the followin operation ***/*

Using a for loop, write all bytes in buffer except for the last one

```
for(i=0; i<num_bytes-1; i++) //num_bytes is 4
{
    i2c_write(buffer[i]);
}
```

Write the last byte and stop

```
i2c_write_and_stop(buffer[num_bytes-1]); //num_by
```

}

Exercise 2: Serial read operation

In the following pseudo-code, interpret the statements and provide the equivalent C-code, using proper syntax. Assume the address of the device is 0x2E and that you are reading 6 bytes of data, starting at register 4 and reading sequentially. On the lines provided, write the C-code. For any function call that requires an argument (variable), indicate the actual value of the variable (except for the data itself).

```
// Read data from the I2C bus
//
// Parameters:
// unsigned char addr { the address of the device to read from
// unsigned char start_reg { the first register to read from
// unsigned char* buffer { array where the read data will be stored
// unsigned char num_bytes { number of bytes to be read from device
void i2c_read_data(unsigned char addr, unsigned char start_reg, unsigned char *buffer,
unsigned char num_bytes)
```

{

Start I2C transfer

```
i2c_start();
```

Write the device address and indicate a write operation on the bus

```
i2c_write(addr & ~0x01); //addr is 0x2E
```

Write the first register to be read and indicate a stop transfer

```
i2c_write_and_stop(start_reg); // start_reg is 4
```

Start I2C transfer

```
i2c_start();
```

Write the device address and indicate a read operation on the bus

```
i2c_write(addr | 0x01); //addr is 0x2E
```

Using a for loop, read all bytes but the last from the slave and store them in the buffer

```
for(j = 0; j < num_bytes - 1; j++) //num_bytes is 6
```

{

AA = 1; *//set acknowledge bit*

```
buffer[j] = i2c_read();
```

// read data

}

AA = 0; *//clear acknowledge bit*

Read the last byte and stop, save it in the buffer

```
buffer[num_bytes - 1] = i2c_read_and_stop(); //num
```

}

When complete, include Worksheet 9 with your Laboratory 3.2 Pre-lab submission.

Code for Data Transfer on I2C Bus

The following functions perform simple operations on the I2C bus. Make sure you understand how each one works. They will be used together to send and receive data over the bus.

```
//-----------------------------------------------------------------------------
// Routine: i2c_read
// Inputs: none
// Outputs: input byte
// Purpose: Reads data from the I2C bus
//-----------------------------------------------------------------------------
unsigned char i2c_read(void)
{
    unsigned char input_data;
    while (!SI);            // Wait until we have data to read
    input_data = SMB0DAT;   // Read the data
    SI = 0;                 // Clear SI
    return input_data;      // Return the read data
}
//-----------------------------------------------------------------------------
// Routine: i2c_read_and_stop
// Inputs: none
// Outputs: input byte
// Purpose: Sends I2C Stop Transfer
//-----------------------------------------------------------------------------
unsigned char i2c_stop_and_read(void)
{
    unsigned char input_data;
    while (!SI);            // Wait until we have data to read
    input_data = SMB0DAT;   // Read the data
    SI = 0;                 // Clear SI
    STO = TRUE;             // Perform I2C stop
    while (!SI);            // wait for stop
    SI = 0;
    return input_data;      //Return the read data
}
//-----------------------------------------------------------------------------
// Routine: i2c_start
// Inputs: none
// Outputs: none
```

```
// Purpose: Sends I2C Start Transfer
//-------------------------------------------------------------------------
void i2c_start(void)
{
    while (BUSY);           // Wait until the SMBus0 is free
    STA = TRUE;             // Set Start bit
    while (!SI);            // Wait until start sent (look at SI)
    STA = FALSE;            // Clear Start bit
    SI = 0;                 // Clear SI
}
//-------------------------------------------------------------------------
// Routine: i2c_write
// Inputs: output byte
// Outputs: none
// Purpose: Writes data over the I2C bus
//-------------------------------------------------------------------------
void i2c_write(unsigned char output_data)
{
    SMB0DAT = output_data; // Store data in SMB0DAT register
    while (!SI);           // Wait until we are done with send
    SI = 0;                // Clear SI
}
//-------------------------------------------------------------------------
// Routine: i2c_stop_and_write
// Inputs: output byte
// Outputs: none
// Purpose: Sends I2C Stop Transfer
//-------------------------------------------------------------------------
void i2c_write_and_stop(unsigned char output_data)
{
    SMB0DAT = output_data; // Store data in SMB0DAT register
    STO = TRUE;            // Set Stop bit
    while (!SI);           // Wait until we are done with send
    SI = 0;                // Clear SI
}
```

Yilu Zhou

EVB Pin		Port Bit	Bit Addresses & Labels
1	2	1.	
		2.	
3	4	3.	
		4.	
5	6	5.	
		6.	
7	8	7.	
		8.	
9	10	9.	
		10.	
11	12	11.	
		12.	
13	14	13.	
		14.	P0.6 — 0x86 SDA
15	16	15.	P0.7 — 0x87 SCL
		16.	
17	18	17.	
		18.	
19	20	19.	
		20.	
21	22	21.	
		22.	
23	24	23.	
		24.	
25	26	25.	
		26.	
27	28	27.	
		28.	
29	30	29.	
		30.	
31	32	31.	
		32.	
33	34	33.	
		34.	
35	36	35.	
		36.	
37	38	37.	
		38.	
39	40	39.	
		40.	
41 ⟷	60		

Software Initializations

A) Port I/0

B) Timers

C) Interrupts

```
EA = 1;
EIE1 |= 0x08;
```

D) A/D

E) PCA

```
PCA0MD =0x81;
PCA0CN = 0x40;
```

F) XBAR

```
XBR0 = 0x27;
```

G) I2C

```
SMB0CR = 0x93;
ENSMB = 1;
```

Z Jin

EVB Pin

1	2
3	4
5	6
7	8
9	10
11	12
13	14
15	16
17	18
19	20
21	22
23	24
25	26
27	28
29	30
31	32
33	34
35	36
37	38
39	40

41 ⟷ 60

	Port Bit	Bit Addresses & Labels
1.		
2.		
3.		
4.		
5.		
6.		
7.		
8.		
9.		
10.		
11.		
12.		
13.		
14.	P0.6	0x86 SDA
15.	P0.7	0x87 SCL
16.		
17.		
18.		
19.		
20.		
21.		
22.		
23.		
24.		
25.		
26.		
27.		
28.		
29.		
30.		
31.		
32.		
33.		
34.		
35.		
36.		
37.		
38.		
39.		
40.		

Software Initializations

A) Port I/0

B) Timers

C) Interrupts

```
EA = 1;
EIE1 |= 0x08;
```

D) A/D

E) PCA

```
PCA0MD =0x81;
PCA0CN = 0x40;
```

F) XBAR

```
XBR0 = 0x27;
```

G) I2C

```
SMB0CR = 0x93;
ENSMB = 1;
```

```
compile derivatives
   #include <studio.h>
   #include <i2c.h>
   #include <c8051_SDCC.h>
   #include <<stdlib.h>
declare global variables
   PCA_COUNTER, PCA_START, READ_COUNTER, DISTANCE
   sbit CF(PCA0 COUNTER OVERFLOW FLAG)
function prototypes
   void SMB_Init(void);
   void PCA_Init(void);
   void XBR0_Init(void);
   void Interrupt_Init(void);
   void PCA_ISR(void) __interrupt 9;
   void Ping_Ranger(void);
   unsigned int Read_Ranger(void);
main function
   declare local variables
      NONE
   initialize system, ports and PCA
      Sys_Init();
      putchar(' ');
      SMB_Init();
      XBR0_Init();
      Interrupt_Init();
      PCA_Init();
   Begin infinite loop
      If 80 ms has passed
         read the ranger using Read_Ranger()
         Set DISTANCE equals the result of Read_Ranger()
         start a ping using Ping_Ranger()
         Reset the READ_COUNTER
         print the range
      End if
   End infinite loop
End main function

functions
   void SMB_Init()
      Set SCL to 100khz
      Enable SMBus
   End SMB_Init()

   void XBR0_Init()
      configure the crossbar the same as the the same as Lab3.1 (0x27)
   End XBR0_Init()

   void Interrupt_Init()
      Enable general Interrupt
      Enable PCA overflow interrupts
   End Interrupt_Init()

   void PCA_Init()
```

```
   Enable SYSCLK/12 and enable interrupts
   Enable PCA COUNTER
End PCA_Init()

void PCA_ISR() __interrupt 9
   Increment PCA_COUNTER to count the number of overflows
   if PCA_COUNTER is larger than 3
      Increment the READ_COUNTER
      Set PCA_COUNTER to 0
   End if
   If PCA interrupt flag is set
      Clear the overflow flag
      Set PCA0 to PCA_START
   End if
   handle other PCA interrupt sources
End PCA_ISR() __interrupt 9

void Ping_Ranger()
   write 0x51 to reg 0 of the ranger
   write one byte of data to reg 0 at addr
End Ping_Ranger()

unsigned int Read_Ranger()
   define local variables
      Data[2]: to store the data from ranger
      range: to store the calculated distance
      addr: the address of the ranger
   read two bytes, starting at reg 2 of the ranger
   calculate the distance from the Data
   return the range
End Read_Ranger()
```

```
compile directives
   #include <studio.h>
   #include <c8051_SDCC.h>
   #include <stdlib.h>
   #include <I2C.h>
declare global variables
   unsigned int PCA_Counter, ReadCompass, PCA_START, h_count, new_heading, CompassD.
   sbit CF (PCA 0 COUNTER OVERFLOW FLAG)
function prototypes

   void SMB_Init(void);
   void PCA_Init(void);
   void XBR0_Init(void);
   void Interrupt_Init(void);
   void PCA_ISR (void) __interrupt 9;

main function
   initialize system, ports and PCA
      Sys_Init();
      putchar(' ');
      SMB_Init();
      XBR0_Init();
      Interrupt_Init();
      PCA_Init();
   print beginning message.
   start while (1) loop
      if 40 ms has passed
         call ReadCompass()
         set CompassD to ReadCompass/10.
         Print CompassD.
   End while (1)loop
End main function

Functions
   void SMB_Init()
      set SCL to 100KHz
      Enable SMBus
   End SMB_Init()

   void XBR0_Init()
      configure the crossbar as directed in the labor manual.
   End XBR0_Init()

   void Interrupt_Init()
      Enable general interrupt
      Enable PCA overflow interrupts
   End Interrupt_Init()

   void PCA_Init()
      Enable SYSCLK/12 and enable interrupt.
      Enable PCA counter
   End PCA_Init()
```

```
void PCA_ISR() __interrupt 9
   Increment PCA_COUNTER to count the number of overflows
   If PCA interrupt flag is set
      Clear the overflow flag
      Set PCA0 to PCA_START
      if two overflow is done
         set h_count to 0
         set new_heading to 1
   handle other PCA interrupt sources
End PCA_ISR() __interrupt 9

unsigned int ReadCompass()
   unsigned char addr to 0xC0 for compass's address
   unsigned char Data[2] that is an array with length of 2
   unsigned int heading for returning degrees between 0 and 3599
   read two byte starting at reg 2
   set heading equals the combine of two values from reg 2
   return heading in thenths degrees
End ReadCompass.
```