1300 Henley Court
Pullman, WA 99163
509.334.6306
www.store.digilent.com

Lab 4d: Communications – I²C Serial Protocols

Revised April 25, 2017
This manual applies to Unit 4, Lab 4d.

1 Objectives

1. Develop a library of functions that manage the functionality and retrieve data from a three-axis accelerometer using I²C communications.
2. Display the accelerometer data on the Basys MX3 LCD.
3. Display the accelerometer data on the workstation monitor using UART asynchronous communications.

2 Basic Knowledge

1. How to exchange serial data with a computer terminal using the PIC32MX370 UART
2. How to use the PIC32MX3270 processor to display text on a character LCD
3. How to implement code reuse that integrates previously developed processor code into new application projects.

3 Equipment List

3.1 Hardware

1. Basys MX3 trainer board
2. Workstation computer running Windows 10 or higher, MAC OS, or Linux
3. 2 Standard USB A to micro-B cables

In addition, we suggest the following instruments:

4. Analog Discovery 2

3.2 Software

The following programs must be installed on your development work station:

1. Microchip MPLAB X® v3.35 or higher
2. PLIB Peripheral Library
3. XC32 Cross Compiler
4. WaveForms 2015 (if using the Analog Discovery 2)
5. PuTTY Terminal Emulation

4 Project Takeaways

1. Understanding of requirements and implementations of synchronous communications.
2. Understanding of the I²C Protocol.
3. Understand the advantages and disadvantages of I²C compared to UART and SPI.

5 Fundamental Concepts

I²C (Inter-integrated Circuit), pronounced *I-squared-C*, is a multi-master, multi-slave, single-ended, serial computer bus invented by Philips Semiconductor (now NXP Semiconductors). It is typically used for attaching lower-speed peripheral ICs to processors and microcontrollers in short-distance, intra-board communication. I²C (sometimes written as I2C) is an example of a synchronous master-slave network supported directly by hardware circuits in many microprocessors. TWI stands for Two Wire interface and, for the most part, this bus is identical to I²C. The name TWI (Two Wire serial Interface) was introduced by Atmel and other companies to avoid conflicts with trademark issues related to I²C. A description of the capabilities of TWI interfaces can be found in the data sheets of corresponding devices. Expect TWI devices to be compatible with the I²C protocol devices, except for some particularities like general broadcast or 10-bit addressing.

I²C will be used in this lab to interface the PIC32MX370 processor with the MMA8652FC 3-Axis Accelerometer.

The I²C protocol uses an open-drain/open-collector with an input buffer on the same line. This allows a single data line to be used for bidirectional data flow. The I²C is a half-duplex scheme where the slave devices are enabled or selected by encoding data in a message sent by the master.

The I²C protocol was developed in the early 1980's by Philips Semiconductors. Its original purpose was to provide an easy way to connect a CPU to peripheral ICs in a TV-set. The original specification supported data rates up to 100 kbits/s. Conventional hardware now supports up to 400 kbits/s.

At any one time, the network consists of only one active master and one or more slave devices. Although slave devices can assert a degree of control over the clock signal, the master is designated as the device that asserts the clock signal indicating when the data signal line should be read by all slave devices, or when a slave device should assert control over the data signal line. Each slave device has a unique device 7-bit identification number. The data is always sent as 8-bit unsigned characters. An I²C message is always initiated with a start signal and terminated with a stop signal that the master controls. The I²C message can be an arbitrarily long stream of data bytes. Each byte of data exchanged between the master and slave device is acknowledged by the receiving master or slave.

Figure 5.1 shows the general format of an I²C message. There are two signals used in the I²C protocol: the serial clock signal (SCL) and the serial data signal (SDA). Other than the start and stop sequences, the SDA is not allowed to change states while the SCL signal is high. When the I²C communications is in an idle state, both the SCL and the SDA signals are in the high state.

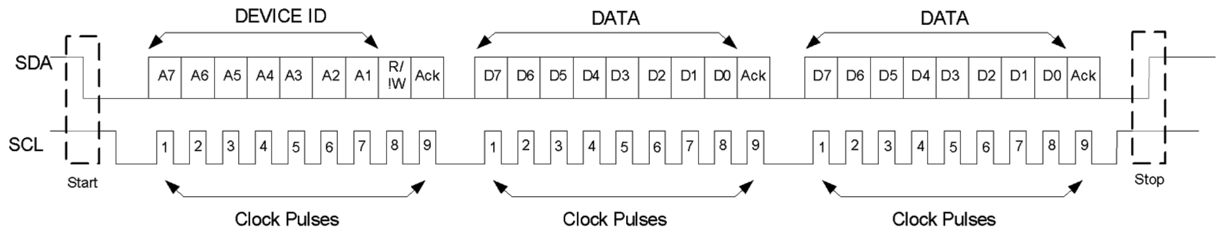

Figure 5.1. Waveform showing the clock and data timing for an I²C message.

The Master always initiates communications with all slave devices that share the SDA and SCL signals by generating a start sequence. The master causes the SDA signal to make a high to low transition while the SCL signal is high. The master must assert the SCL signal low before the process of sending data begins. Data bits are sent by asserting the SDA signal high or low followed by asserting the SCL signal high for a specified interval of time. The most significant data bit is sent first.

The master always sends the first 8 data bits that consists of a 7-bit slave device address, identified as A7 through A1 in Fig. 5.1, and a control bit, R/!W, that specifies the direction of data flow for successive communications. The R/!W bit is low if subsequent data is to be written from the master to the slave device. If the R/!W bit is high, then subsequent bits of data are read from the slave.

Each slave device has, within the device's hardware, a unique identification number. The device that has the identification number that matches the device address field sent from the master will acknowledge the first byte in the I²C message with an ACK bit. The microprocessor serving as the I²C master will terminate communications if no acknowledge is generated after any 8-bit data transfer from the master to the slave. The communications message is terminated by generating a stop sequence. A stop sequence occurs when the SDA signal is set to the high state, followed by setting the SCL signal to a high state. Normally both the SDA and SCL are left in the high state during idle periods.

5.1 I²C PHYSICAL Layer

I²C networks consist of a data signal (SDA) and a clock signal (SCL) that have a common reference – usually digital ground. A pull-up resistor to V_{DD} is required to be connected to each signal line. Both clock and data signals are connected in a wired-AND configuration that require open collector (also called open drain for CMOS transistors) outputs from both master and slave devices. The output of both devices must be in the open collector state for the signal line to be in the high state (also called the recessive state). The SDA or SCL line is low if the output transistor of either the master or any slave device pulls the signal low. The low state is also referred to as the dominant state. As shown in Fig. 5.2, both slave and master devices can always determine the state of the SCL or SDA lines by reading the value of the output pin.

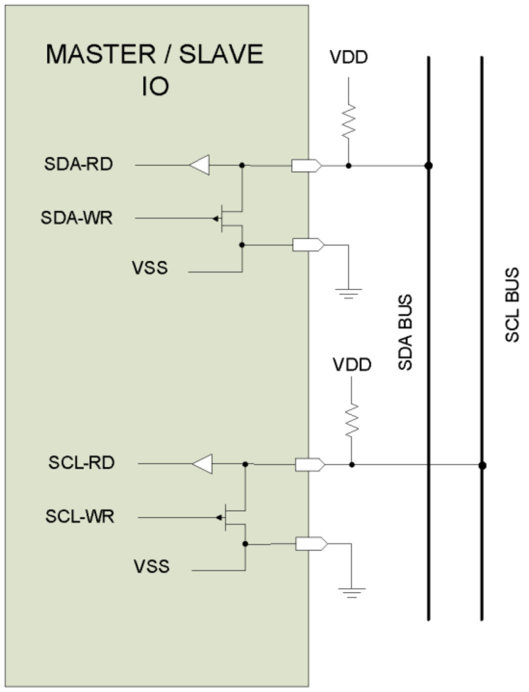

Figure 5.2. Circuit diagram of I²C device pins.

As stated above, the SDA should not change levels while the SCL is high except to implement the START and STOP sequences. As illustrated in Fig. 5.3, a start sequence is generated when the SDA line makes a recessive to dominant transition when the SCL line is in the recessive state. Figure 5.4 shows that a STOP sequence is generated when the SDA signal transitions from the dominant state to the recessive state while the SCL line is in the recessive state. Figure 5.5 shows that data is either written to the slave or read from the slave during the period identified as 2 to 3. The master or slave (depending on whether it is a write or read operation) is allowed to change the state of the SDA line while the SCL line is in the dominant state, identified as 1 and 4 in Fig. 5.5. Figure 5.6 shows an acknowledge sequence that is generated by the slave after the master writes a byte of data, or is generated by the master after the master reads a byte of data from the slave.

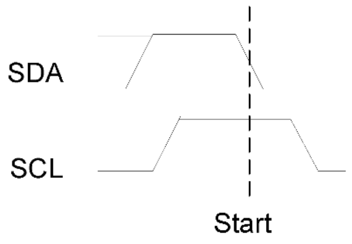

Figure 5.3. I²C START condition.

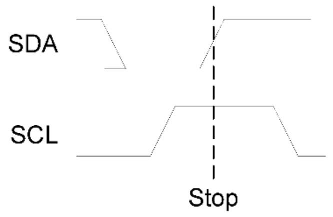

Figure 5.4. I²C STOP condition.

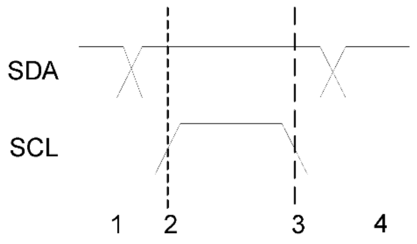

Figure 5.5. I²C data read/write sequence.

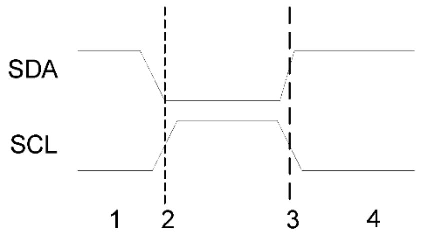

Figure 5.6. I²C acknowledge sequence.

The *acknowledge* (ACK) bit is generated by the SDA being in the dominant state when the SCL is asserted to the recessive state during the ninth clock cycle. The ACK always follows the 8 data bits. The slave device must always ACK the first byte in an I²C message which contains both the device identification and the R/W bit. The master device must assert the ACK signal when receiving data from the slave device indicating the data has been received to the slave device.

No ACK is generated if the SDA line is not pulled to the dominant state by either the master or the slave during the ninth clock pulse. There are two uses of a no ACK condition. When a master intends to terminate read sequence, it will not generate an ACK. This is called a NACK condition and does not constitute an error condition. The second use of a *NO ACK* is when the slave fails to pull the SDA to the dominant state during the ninth SCL pulse. This *is* an error condition and is detectable by the master. Possible reasons for a *NO ACK* condition are: the slave is not connected to the I²C lines, the incorrect device address was sent by the master, or that the slave device is not functional.

5.2 Master-Slave I²C Network Architecture

Figure 5.7 shows a conventional I²C network connection between the master and one or more slave devices. The architecture is classified as a bus network because all devices on the network share the same physical communications medium.

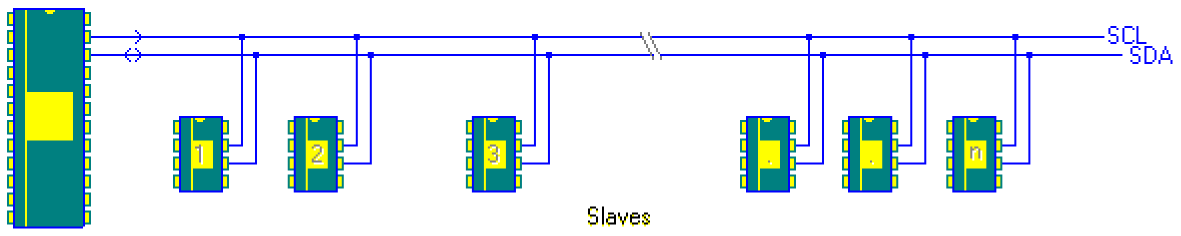

Figure 5.7. Single master I²C network architecture.

Slower slave devices can synchronize high speed masters by asserting control over the SCL signal using clock-stretching. The master generates its own clock on the SCL line to transfer messages on the I²C bus. Data is only valid during the HIGH period of the clock. A well-defined clock is therefore needed for the bit-by-bit arbitration procedure to take place.

I²C clock synchronization by clock-stretching, as shown in Fig. 5.8, is performed using the wired-AND connection of I²C interfaces to the SCL line. As soon as the master asserts the SCL line in the recessive state, a slave device that wants to slow the master down simply holds the SCL line in the dominant state until the slave determines when to release the SCL line to the recessive state. Meanwhile, the master always reads the SCL IO pin and doesn't start the I²C clock cycle until it detects the SCL line in the recessive state. Thus the slowest device on the I²C network is able to dictate the maximum data transfer rate. The one caution concerning the use of clock-stretching by the slave I²C device is that this can constitute software blocking unless some form of timeout is implemented by the I²C bus master.

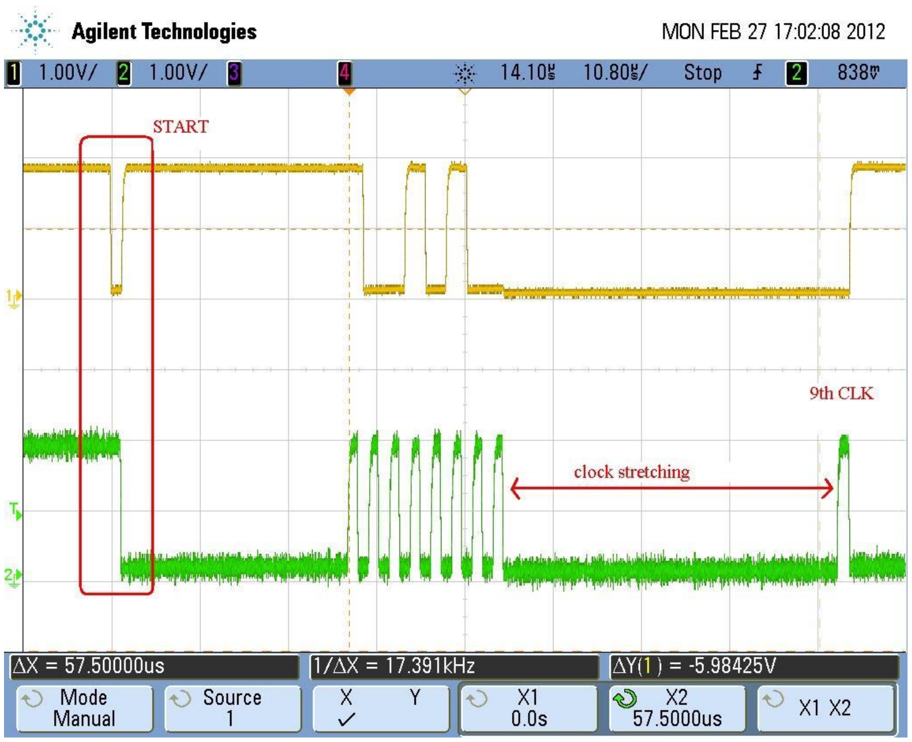

Figure 5.8. Waveforms of SDA and SCL demonstrating clock-stretching.

5.2 Multi Master I²C Network

I²C technology allows for multiple masters to operate on a single I²C network as shown in Fig. 5.9. As stated previously, the controlling master asserts the state of the SCL line. Hence, the master that is ***not*** controlling the SCL line must behave like a slave device. Each master must be able to implement an arbitration scheme that dictates that if two devices start to communicate at the same time, the one writing more zeros (dominant bits) to the bus wins the arbitration and the other master device immediately discontinues any operation on the bus.

The second requirement is that each master device must be able to detect when the network is in use by another master. Each potential device must detect an ongoing bus communication and must not interfere with it. This is achieved by recognizing traffic and waiting for a stop condition to appear before starting to transmit on the bus. Fig. 5.10 shows a timing diagram where two master devices attempt to simultaneously access the same slave device. The SCL signals will be automatically synchronized according to the process described above for slow slave devices. A bus conflict will be detected by the master device that attempts to send a recessive bit but detects the SDA line in the dominant state, as shown for CPU2. At this point, CPU2 places its SCL and SDA outputs in the recessive state and continues to monitor the SCL line. After there has been no activity on the SCL line for a predetermined length of time, CPU2 will attempt to assume control of the I²C bus once again.

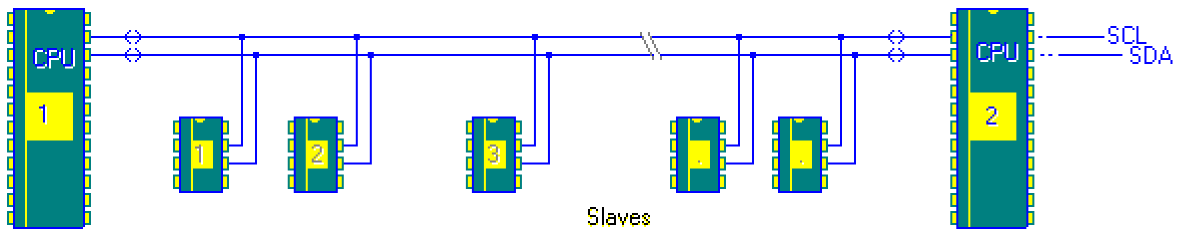

Figure 5.9. Multiple master I²C network architecture.

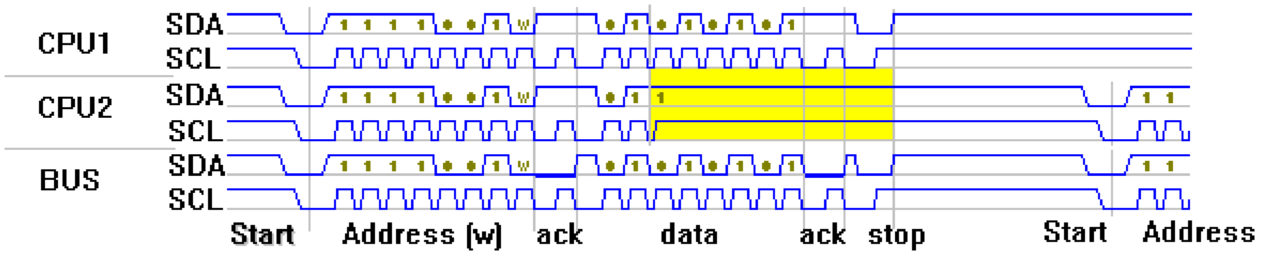

Figure 5.10. Clock arbitration for dual master I²C operation.

5.4 I²C on the PIC32MX370 Processor

The PIC32MX370 processor can simultaneously support two independent I²C channels at data rates up to 1 MBaud. The SCL and SDA pins for I²C channel 1 are permanently assigned to pins 2 and 3 of IO port G. The SDA and SCL pins of I²C channel 1 are connected to the on-board MMA8652FA accelerometer along with the necessary pull-up resistors. The SCL and SDA signals are also routed to J7 to allow external devices to be connected to the I²C bus.

Though the PIC32MX processor can implement 7-bit addressing as described above, or 10-bit addressing, the 7-bit addressing mode is most common. (The waveform shown in Fig. 5.10 actually represents a 10-bit address.) Since the device address 0x78 through 0x7B are reserved for the 10-bit addressing mode, only 123 unique device addresses are possible. If the occasion arises that two I²C slave devices have the same device address, it is possible to use both of the I²C channels with one device assigned to one channel and the other to the second channel.

5.5 PIC32 I²C Hardware

Figure A.2 shown in Appendix A is a block diagram of the I²C hardware in the PIC32MX processor. The PIC32MX microprocessor can be used as an I²C Master or an I²C Slave. Obviously the microprocessor internal hardware to implement the I²C protocol is complex, and as will be shown below, software to manage all aspects of an I²C port is involved. The I²C hardware and protocol is primarily intended for short distances like one would see on a printed circuit board. The maximum data rate is dependent on a number of physical characteristics of the cabling between master and slaves. In order to have extended length I²C busses, one must operate at reduced clock speeds. Since I²C uses a synchronizing clock, there is no problem operating I²C slaves at reduced clock speeds. References 4 and 5 provide an in depth treatment on this subject.

5.6 PIC32 I²C Software

It is suggested that the reader open the MPLAB X Help window and open the XC32 -> XC32 Peripheral Library –> I²C option.

Before the I²C channel can be used, it must be initialized. The I²C_init function shown in Listing B.1 of Appendix B can be used for either I2C1 or I2C2 channels and at a specified bit rate. If the actual I²C clock frequency is not within 10% of the specified I²C clock frequency, the error is reported as a text message to the UART. Otherwise, the I²C port is enabled.

The I²C master *read* and *write* operations can be managed by four primitive functions consisting of Start Transfer, Write One Byte, Read One Byte, and Stop Transfer. Listing B.2 through Listing B.5 in Appendix B provide the C language code for these four functions.

6 Problem Statement

The PIC32MX370 processor will periodically read the physical orientation data from a 3-Axis accelerometer and display the information on a character LCD and send a text stream to a computer terminal.

7 Background Information

The following discussion is not intended to be a complete software guide to the operations and uses for the MMA8652FC accelerometer. It is intended to demonstrate an application of the I²C communications protocol. Reference 3 and 4 listed at the end of this document provide more complete user information. The legend on the processor board as shown in Fig. 7.1 provides a reference for the orientation of the accelerator IC. The schematic diagram in Fig. 7.2 shows that the SDA and SCL have 2.2k ohm pull-up resistors installed. Hence, there is no need for external pull-up resistors if an external I²C device is attached to the processor board.

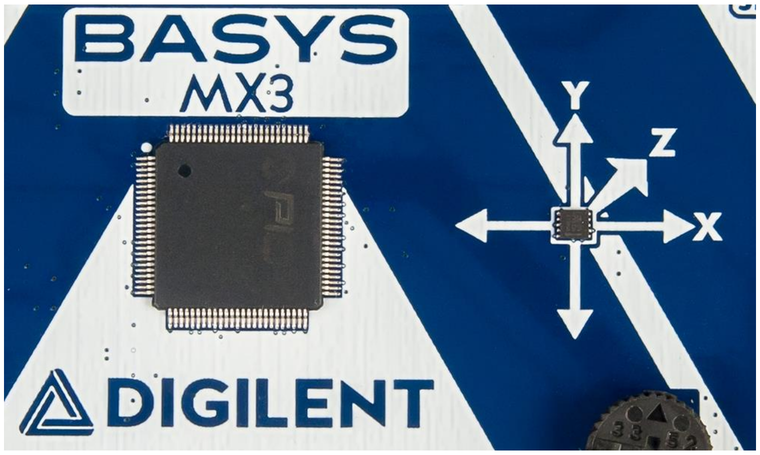

Figure 7.1. Picture of the MMA8652FC Accelerometer mounted on the Basys MX3 processor board.

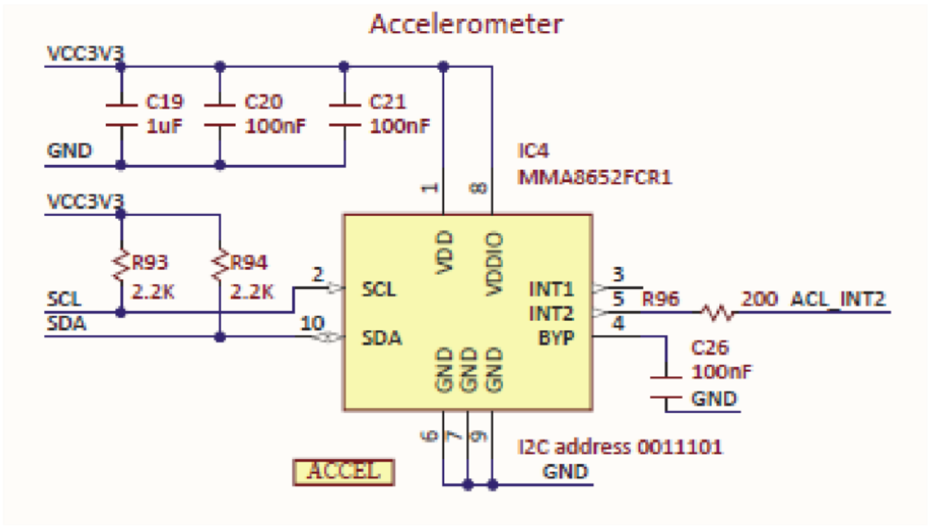

Figure 7.2. Schematic diagram of MMA8652FC accelerator on the Basys MX3 processor board.

7.1 The MMA8652FC 3-AXIS Accelerometer Software

The communications with the accelerometer requires two global functions that are made up of calls to the four hardware interface functions shown in Listings B.6 through B.9. The control flow diagrams for writing and reading a succession of accelerometer registers are shown in Fig. 7.3 and Fig. 7.4. These two functions can only be called after the accelerometer has been initialized using the code in Listing B.5. The shaded blocks in Fig. 7.3 and Fig. 7.4 refer to coded functions provided in Listings B.2 through B.6. With the aid of the control flow diagrams shown in Fig. 7.3 and Fig. 7.4, the user is expected to develop their one i2cWriteBlk and i2cReradBlk functions.

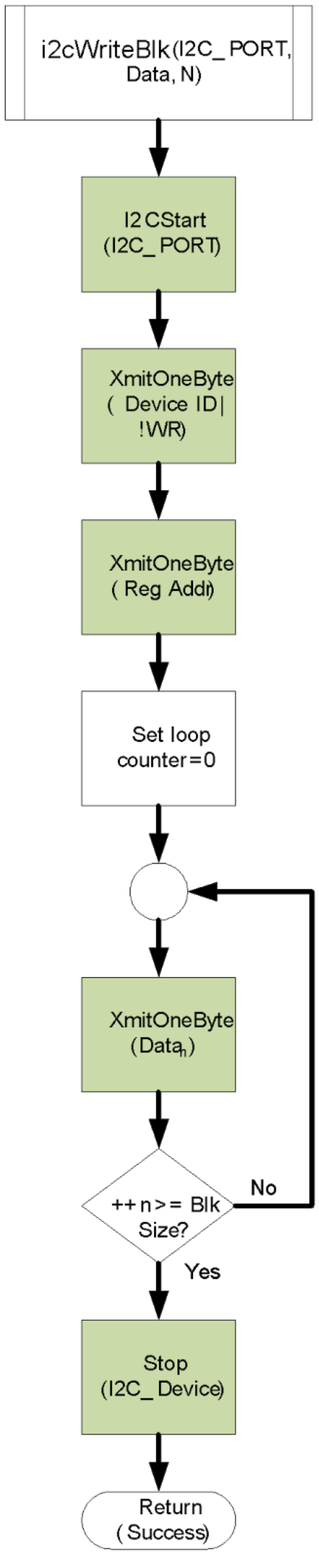

Figure 7.3. Control Flow Diagram for writing a block of data to the MMA8652FC Accelerometer.

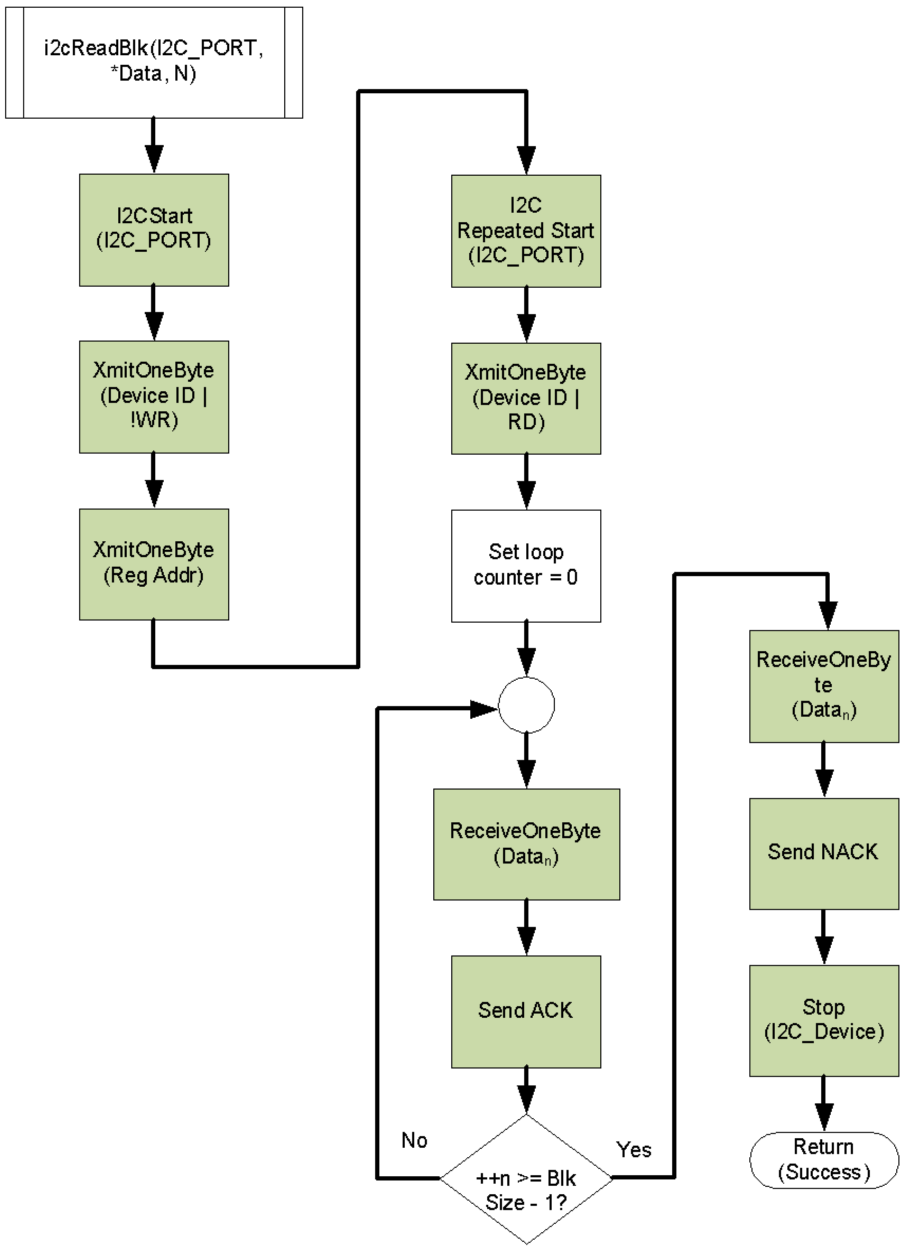

Figure 7.4. Control Flow Diagram for reading a block of data from the MMA8652FC Accelerometer.

Initialization of the MMA8652FC accelerometer, as shown in Reference 3, consists of placing the device in standby mode, setting the operating modes, and placing the device in Active mode. Since a block of data can be any number of bytes (zero to N), the I²C_Read_Blk and I²C_Write_Blk functions are all that are needed to interface with the accelerometer.

8 Lab 4d

8.1 Requirements

1. The PIC32MX370 UART channel 4 must operate at 38400 BAUD with no parity.
2. Initialize the MMA8652FC 3-Axis Accelerometer as follows:

   a. Set to standby so device can be configured
   b. Set to 2G Full scale mode
   c. Set F-Mode to 0
   d. Set back to Active mode
3. Read the X, Y and Z-axis data once each second.
4. Display data from all three axis on the LCD using the format for the first line "X:#### Y:####" and for the second line "Z:####" once each second.
5. Display data from all three axis on the LCD using the format "X:#### Y:#### Z:####" once each second.

8.2 Design Phase

1. Develop a data flow diagram for the software components needed for the requirements of Lab 4d.
2. Schematic diagrams: Provide a block diagram of the equipment used for Lab 4d.
3. Flow diagrams: Provide a complete software control flow diagram for lab 4d.

8.3 Construction Phase

1. Create a new project named Lab4d.
2. Add C program files for configuring the processor and initializing the IO for the switches and LEDs on Basys MX3 board.
3. Add the C program files developed for previous labs that provide interfaces for the LCD and UART.
4. Develop a file containing the I^{2}C functions listed Appendix B.
5. Add the functions for i2cWriteBlk and i2cReadBlock following the control flow diagrams shown in Fig. 7.3 and Fig. 7.4.
6. Download the completed functional project to the PIC32MX370 processor.
7. Unplug the Analog Discovery 2 during testing to make it easier to position the Basys MX3 board.

8.4 Testing

1. Connect the Basys MX3 UART to the workstation and launch the workstation's terminal emulation program.
2. Place the Basys MX3 board in a position that maximizes the X-axis value and minimizes the Y and Z-axis values. Note the edge of the Basys MX3 board is in the horizontal position.
3. Place the Basys MX3 board in a position that maximizes the Y-axis value and minimizes the X and Z axis values. Note the edge of the Basys MX3 board is in the horizontal position.
4. Place the Basys MX3 board in a position that maximizes the Z-axis value and minimizes the X and Y-axis values. Note the edge of the Basys MX3 board is in the horizontal position.
5. Using the Analog Discovery 2, capture a screen that shows the I^{2}C SCL and SDA signals when the PIC32 is reading data from all three axes.

9 Questions

1. Using the waveform captured in step 5 of the testing section, identify all portions of the I^{2}C SDA and SCL signals starting with the START sequence and ending with the STOP sequence.
2. Compute the effective data rate in bits per second.
3. What effect, if any, would a change in the pull up resistors value for the SDA and SCL have on the I^{2}C bus?
4. Should there be current (e.g., maximum amperage) specifications associated with these protocols?

5. Identify as many things about the circuitry that will limit the maximum rate at which data can be reliably transmitted, and explain why (e.g., what effect might the length of the wires have?).
6. What effect might capacitance between data transmission and ground have? Is it possible that the power supply could have an effect, and if so, what?
7. Explain why you may use I²C, a serial protocol over SPI, another serial protocol.
8. What are the strong points of I²C and RS232 protocols compared with each other?
9. Describe the Basys MX3 board orientation noted in the three testing assignments.

10 References

1. Embedded Computing and Mechatronics with the PIC32 Microcontroller, 1st Edition, by Kevin Lynch (Author), Nicholas Marchuk (Author), Matthew Elwin (Author), https://www.amazon.com/Embedded-Computing-Mechatronics-PIC32-Microcontroller/dp/0124201652
2. "PIC32MX330/350/370/430/4450/470 32 Bit Microcontroller Datasheet (60001185E)", http://ww1.microchip.com/downloads/en/DeviceDoc/60001185E.pdf
3. Understanding the I²C Bus, http://www.ti.com/lit/an/slva704/slva704.pdf
4. Design calculations for robust I2C communications, http://www.edn.com/design/analog/4371297/Design-calculations-for-robust-I2C-communications
5. I²C Maximum Clock Speed Calculator, http://interfacechips.eeweb.com/wp-content/uploads/2014/02/i2c.clock_.speed_.calculator.pdf
6. MMA8652FC, 3-Axis, 12-bit, Digital Accelerometer Data Sheet, Document Number: MMA8652FC, Rev. 3.3, 10/2015, http://cache.freescale.com/files/sensors/doc/data_sheet/MMA8652FC.pdf
7. "Data Manipulation and Basic Settings for Xtrinsic MMA865xFC Accelerometers", Fengyi Li, http://cache.freescale.com/files/sensors/doc/data_sheet/AN4083.pdf

Appendix A: Unit 4 Parts Configuration

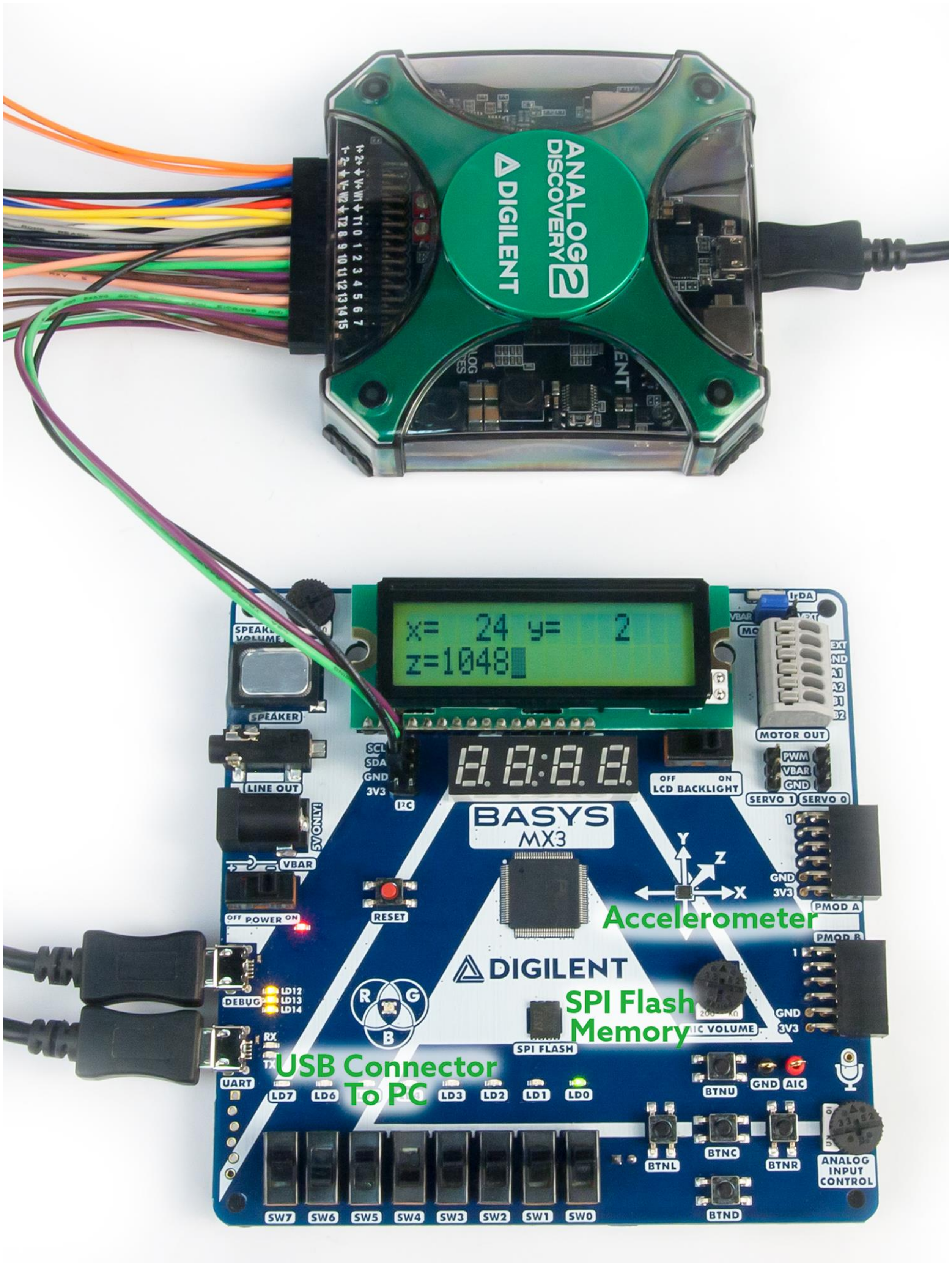

Figure A.1. Unit 4 hardware and instrumentation configuration.

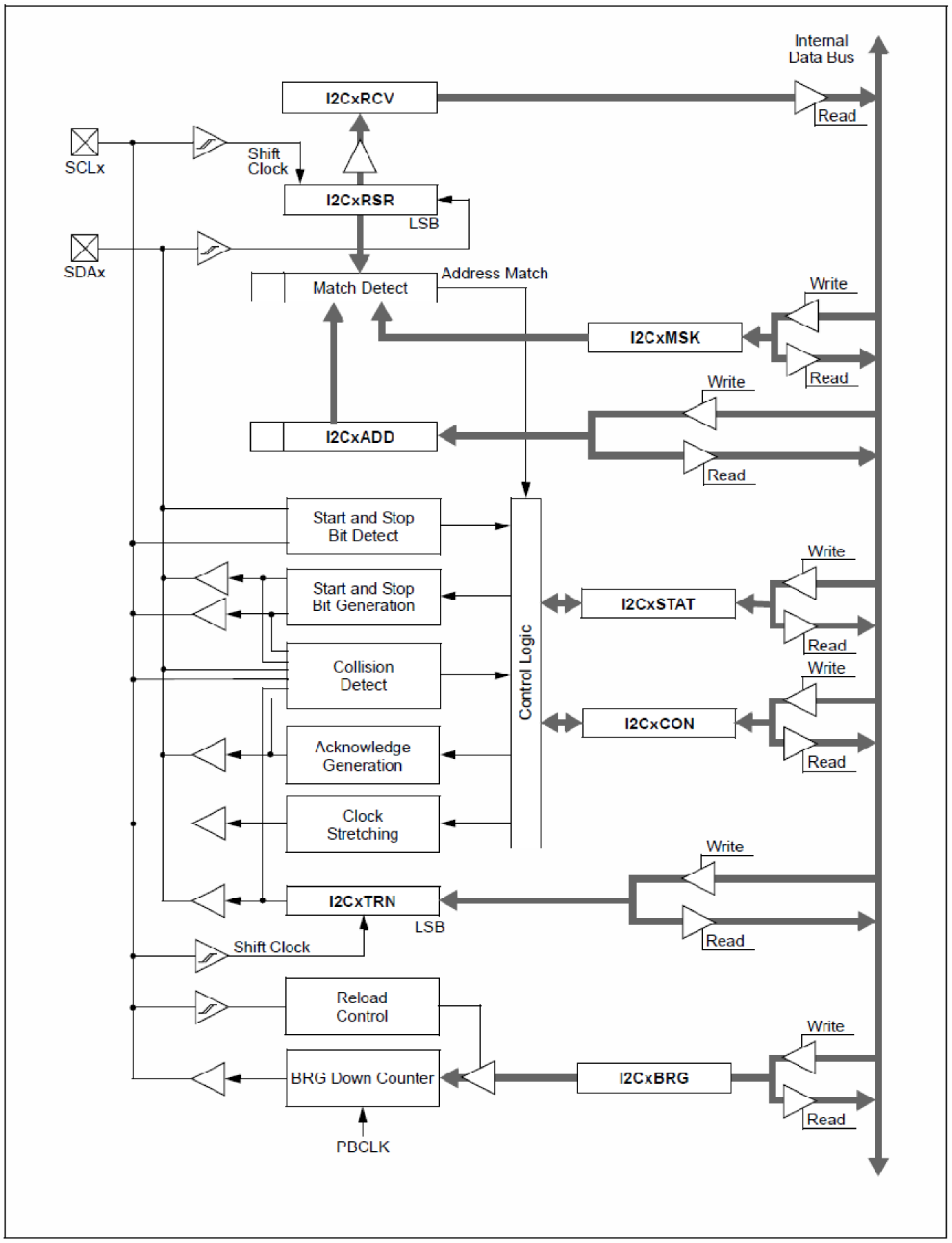

Figure A.2. PIC32MX I²C hardware block diagram.

Appendix B: Allocating a Heap in MPLAB X

Listing B.1. I²C Master Initialization Function

```
BOOL i2cInit(I2C_MODULE i2cPort, unsigned int i2cClk)
{
unsigned int actualClock;
BOOL success;
/* Set Desired Operation Frequency */
   actualClock = I2CSetFrequency(i2CPcort,GetPeripheralClock(),i2cClk);
   if ( abs(actualClock-bitrate) > bitrate/10 )
   {
       printf("  Clock frequency (%d) error exceeds 10\%\r\n",
                (unsigned) actualClock);
       success = FALSE;
   }
   else
   {
       printf(" I2C%d clock frequency = %ld Hz\n", i2cPort+1,
                (unsigned int) actualClock);
       success = TRUE;
   }

   I2CEnable(i2cPort, TRUE);
   return success;
} /* End of i2cInit */
```

Listing B.2. I²C Master Start Transfer Function

```
static BOOL i2cStartTransfer( I2C_MODULE i2cPort, BOOL restart )
{
I2C_STATUS  status;


/* Send the Start (or Restart) sequence */
   if(restart)
   {
       I2CRepeatStart(i2cPort);
   }
   else
   {
/* Wait for the bus to be idle, then start the transfer */
       while( !I2CBusIsIdle(i2cPort) );


       if(I2CStart(i2cPort) != I2C_SUCCESS)
       {
           printf("Error: Bus collision during transfer Start\n");
           return FALSE;
       }
   }


/* Wait for the START or REPEATED START to complete */
   do
   {
       status = I2CGetStatus(i2cPort);
   } while ( !(status & I2C_START) );
   return TRUE;
} /* End of i2cStartTransfer */
```

Listing B.3. I²C Master Start Transfer Function

```
static void i2cStopTransfer( I2C_MODULE i2cPort )
{
I2C_STATUS  i2cStatus;


/* Send the STOP sequence */
   I2CStop(i2cPort);


/* Wait for the STOP sequence to complete */
   do
   {
       i2cStatus = I2CGetStatus(i2cPort);


   } while ( !( i2cStatus & I2C_STOP) );
} /* End of i2cStopTransfer */
```

Listing B.5. I²C Master Transmit One Byte Function

```
static BOOL i2cTransmitOneByte( I2C_MODULE i2cPort, BYTE data )
{
/* Wait for the transmitter to be ready */
   while(!I2CTransmitterIsReady(i2cPort));
/* Transmit the byte */
   if(I2CSendByte(i2cPort, data) == I2C_MASTER_BUS_COLLISION)
   {
       printf("Error: I2C Master Bus Collision\n");
       return FALSE;
   }
/* Wait for the transmission to finish */
   while(!I2CTransmissionHasCompleted(i2cPort));
   return TRUE;
} /* End of i2cTransmitOneByte */
```

Listing B.6. I²C Master Transmit One Byte Function

```
static I2C_RESULT i2cReceiveOneByte( I2C_MODULE i2cPort, BYTE *data, BOOL ack )
{
I2C_RESULT i2cOps = I2C_SUCCESS;


   if(I2CReceiverEnable(i2cPort, TRUE) == I2C_RECEIVE_OVERFLOW)
   {
       printf("Error: I²C Receive Overflow\n");
       i2cOps =  I2C_RECEIVE_OVERFLOW;
   }
   else
   {
       while(!I2CReceivedDataIsAvailable(i2cPort));
/* The "ack" parameter determines if the slave READ is acknowledged */
       I2CAcknowledgeByte(i2cPort, ack);
       while(!I2CAcknowledgeHasCompleted(i2cPort));
/* Read the received data byte */
       *data = I2CGetByte(i2cPort);
   }
   return i2cOps;
} /* End of i2cReceiveOneByte */
```