Electrical and Electronic Engineering 2019, 9(2): 45-52
DOI: 10.5923/j.eee.20190902.04

PIC32MZ Custom I2C Master Communication API Library

Abhishek N. Patel

Research and Development Department, Horiba Instruments Inc., 20 Knightsbridge Road, Piscataway, New Jersey, United States

Abstract This paper demonstrates how to make I2C master communication library routines for PIC32MZ/PIC32 MCUs. I2C is a synchronous serial communication protocol. It provides the good support for communication with slow peripheral devices such as EEPROM, ADC, RTC, Voltage Monitor etc. Unfortunately, the microchip PICXC32 compiler does not gives us the I2C library APIs to call from firmware application layer. This implementation requires PIC32MZ/PIC32 MCU based hardware and microchip MPLAB-X IDE, tool chain and Harmony framework.

Keywords C, PIC32MZ, PIC32, Firmware, Embedded System, i2C

1. Introduction

The I2C-bus developed by Philips to maximize hardware efficiency and circuit simplicity. The I2C interface is a simple master/slave type interface. In I2C, both buses are bidirectional, which means master able to send and receive the data from the slave. I2C is a 2-wire design, one wire used for serial data line (SDA) and other one used for serial clock line (SCL). I2C is appropriate for interfacing to devices on a single board. This can be stretch across multiple boards inside a closed system, but not much further.

For an example, the PIC MCU on a main embedded board using I2C to communicate with user interface devices located on a separate front panel board. A second example is SDRAM DIMMs, which can feature an I2C EEPROM containing parameters needs to correctly configured a memory controller for that module.

2. I2C Detail

The two I2C signals are serial data (SDA) and serial clock (SCL). Together, these signals make it possible to support serial.

Transmission of 8-bit bytes of data-7-bit device addresses and control bits-over the two-wire serial bus. The lines are open-drain (open-collector). This means that devices on the I2C bus can only pull the line low or leave them open/floating. This means that both SDA and SCL needs a pull-up resistor, otherwise they will not be able to set the lines high.

The device that initiates a transaction on the I2C bus is termed the master. The master normally controls the clock signal. A device being addressed by the master is called a slave.

In a bind, an I2C slave can hold off the master in the middle of a transaction using what via clock stretching (the slave keeps SCL pulled low until it is ready to continue). Most I2C slave devices do not use this feature, but every master should support it.

The I2C protocol supports multiple masters, but most system designs include only one. There may be one or more slaves on the bus. Both masters and slaves can receive and transmit data bytes.

Each I2C-compatible hardware slave device comes with a predefined device address, the lower bits of which may be configurable at the board level. The master transmits the device address of the intended slave at the beginning of every transaction. Each slave is responsible for monitoring the bus and responding only to its own address.

2.1. I2C Protocol Overview

Both SDA and SCL are bidirectional communication signals and can be change by either master or slave. SDA is the Serial data line, and SCL is the Serial clock line. Every device hooked up to the bus has its own unique address, no matter whether it is an MCU, LCD driver, EEPROM memory, or ASIC. Each of these chips can act as a receiver and/or transmitter, depending on the functionality. Obviously, an LCD driver is only a receiver, while an EEPROM memory or I/O chip can be both transmitter and receiver.

The I2C bus is a multi-master bus. This means that more than one IC capable of initiating a data transfer and could be

* Corresponding author:
a_n_patel@yahoo.com (Abhishek N. Patel)
Published online at http://journal.sapub.org/eee

connected to it. The I2C protocol specification states that the IC that initiates a data transfer on the bus and considered the Bus Master, which generally is a microcontroller (PIC32MZ for this article). Consequently, at that time, all the other ICs considered to be as a Bus Slaves.

First, the MCU will issue a START condition. This acts as an 'Attention' signal to all of the connected devices. All ICs on the bus will listen to the bus for incoming data.

Then the MCU sends the ADDRESS of the device it wants to access, along with an indication whether the access is a Read or Write operation. Having received the address, all IC's will compare it with their own address. If it does not match, they simply wait until the bus is released by the stop condition. If the address matches, however, the chip will produce a response called the ACKNOWLEDGEMENT signal.

Once the MCU receives the acknowledgement, it can start transmitting or receiving DATA. When it complete the trasaction, the MCU will issue the STOP condition. This is a signal that the bus has been release and that the connected ICs may expect another transmission (DATA transaction) to start any moment.

2.2. I2C Order of Operation

Unlike UARTs and SPI, I2C employs a handshaking protocol between master and slave. Let us look at two example. In the first, a byte (8-bits) of data is being written to a slave that has an I2C address of 0x50. The slave's internal register number is 0x55 and we want to set it to a value of 0x11.

The order of operations would be:

- Send start signal
- Write slave address with Read/Write bit set to 0 (Send 0x50 << 1 = 0xA0)
- Receive ACK
- Write register address (0x55)
- Receive ACK
- Write data value (0x11)
- Send stop signal

That second line is going to be confusing. Take a look again at how the 9-bit number is made up above. We are actually only sending 8 bits, the ACK is automatic so let's look at the first 8 bits. The upper 7 bits contain the slave's address and the least significant bit is the R/W bit. This means that 7-bit I2C addresses need to be shifted left by 1 when writing them to the I2C bus.

For the second example, 8 bits of data will be read from the same slave, register number 0x72.

The order of operations would be:

- Send start signal
- Write slave address with Read/Write bit set to 1 (Send 0x50 << 1 | 1 = 0xA1)
- Receive ACK
- Write register address (0x72)
- Receive ACK
- Receive data byte (8-bits)
- Send NACK
- Send stop signal

Again, that second line. If we want to set the least significant bit, we can either add 1 to the address or "OR" it by 1, which I prefer because it looks more confusing. To receive 8-bit data, send NACK instead of ACK. If we send an ACK, the slave device will start sending me more data.

3. Detail of PIC32MZ I2C Master API/Routines

The 144-pin PIC32MZ device has five I2C peripherals, which control through seven SFRs. Many of the fields in these SFRs initiate an "event" on the I2C bus. All bits default to zero except I2CxCON.SCLREL. In the SFRs below, x refers to the I2C peripheral number, 1, 3, 4, or 5. The following registers are used:

- I2CxCON - I2Cx Control Register - Used to set up I2C peripheral "x"
- I2CxBRG - I2Cx Baud Rate Generator Register - Used for setting the speed of the I2C peripheral "x"
- I2CxTRN - I2Cx Transmit Data Register - Contains data we want to send onto the I2C bus for peripheral "x"
- I2CxRCV - I2Cx Receive Data Register - Contains data received from the I2C bus for peripheral "x"
- I2CxADD - I2Cx Peripheral Address Register - Contains I2Cx peripheral slave address (Only for the Slave Mode).
- I2CxSTAT - I2Cx Status Register - Contains the status of the I2C peripheral "x"

For example to configure I2C peripheral "1" as a master, we need to consider I2C1Con, I2C1BRG, I2C1TRN, I2C1RCV and I2C1STAT registers. This means the two physical pins we will connect to are SDA1 (on port RA15) and SCL1 (on port RA14).

3.1. Configure the Speed of the I2C Peripheral 1 via I2C1BRG Register

As per the PIC32MZ product page we can run from 100kHz to 1Mhz, the lower 16 bits of I2CxBRG register determine the baud. Typically, the baud is either 100 kHz or 400 kHz. To compute the value of I2CxBRG, use the formula:

$$I2CxBRG = \left[\left(\frac{1}{(2 \cdot FSCK)} - TPGD\right) \cdot PBCLK\right] - 2$$

Where Fsck is the desired baud (i.e. 100kHz), Fpb is the peripheral bus clock frequency, and TPGD is 104 ns, according to the Reference Manual. With a 100MHz peripheral bus clock, I2C1BRG = 487 for 100 kHz. Only masters need to set a baud rate.

Implement the code to do that:

```
// Setup the I2C Master
void i2c_master_setup(void)
{
        I2C1BRG = 487;  // I2CBRG = [1/(2*Fsck) - PGD]*Pblck - 2
                        // Fsck is the freq (100 kHz here), PGD =
104 ns
                        // Pblck is Peripheral clock freq (100 MHz)
        I2C1CONbits.ON = 1;     // turn on the I2C1 module
}
```

3.2. Create I2C Master Start, Stop, Restart, ACK and NACK API

The implementation of the I2C master contains functions roughly corresponding to the primitives discussed earlier. Each function executes the primitive command and waits for it to complete. By calling the primitive functions in succession, you can form an I2C transaction. The I2C1CON register mainly used to perform all of these:

- I2C1CONbits.SEN - Start Condition Enable bit
- I2C1CONbits.PEN - Stop Condition Enable bit
- I2C1CONbits.RSEN - Restart (Repeated start) Condition Enable bit
- I2C1CONbits.ACKDT - Acknowledge Data bit. Set to 0 to ACK and 1 for NACK.
- I2C1CONbits.ACKEN - Acknowledge Sequence Enable bit

The Library API routine code:

```
// Start a transmission on the I2C bus.
void i2c_master_start(void)
{
     i2c_master_idle();          // Check for Bus Status.
     I2C1CONbits.SEN = 1;        // send the start bit

    // wait for the start bit to be sent
     while(I2C1CONbits.SEN) { ; }
}

// Stop a transmission on the I2C bus.
void i2c_master_stop(void)
{
     i2c_master_idle();        // Check for Bus Status.
    // comm is complete and master relinquishes bus
    I2C1CONbits.PEN = 1;
     // wait for STOP to complete
     while(I2C1CONbits.PEN) { ; }
}

// Restart a transmission on the I2C bus.
void i2c_master_restart(void)
{
       i2c_master_idle();             // Check for Bus Status.
        I2C1CONbits.RSEN = 1;   // send a restart
       // wait for the restart to clear
        while(I2C1CONbits.RSEN) { ; }
}

// Check the bus status
void i2c_master_idle(void)
{
    // Acknowledge sequence not in progress
    // Receive sequence not in progress
    // Stop condition not in progress
    // Repeated Start condition not in progress
    // Start condition not in progress
     while(I2C1CON & 0x1F);

     // Bit = 0 ? Master transmit is not in progress
     while(I2C1STATbits.TRSTAT);

     return;
}

// Send an acknowledge to the Slave.
void i2c_master_ack_nack(int val)
{
    // To sends ACK        = 0     (slave should send another byte)
   //     or   NACK    = 1 (no more bytes requested from   slave)

      i2c_master_idle();    // Check for Bus Status.
     // store ACK/NACK in ACKDT
     // send ACKDT
     // wait for ACK/NACK to be sent
     I2C1CONbits.ACKDT = val;
     I2C1CONbits.ACKEN = 1;
     while(I2C1CONbits.ACKEN) { ; }
}
```

3.3. Create I2C Master Send and Receive Data API

The I2C1TRN register handles sending data functionality, which is a matter of writing to it and waiting for the Transmit Buffer to be empty by checking Transmit Buffer Full (TBF) flag. After that waiting for the slave device to Acknowledge receipt.

For receiving data, wait for it to clear the flag and until the receive buffer is full by checking Receive Buffer Full (RBF) flag and then send either an ACK or a NACK.

```
// Send one byte data to the slave and check the acknowledged.
BOOL i2c_master_send(unsigned char byte)
{
     // send a byte to slave
    I2C1TRN = byte;   // if an address, bit 0 = 0 for write, 1 for read
   // wait for the transmission to finish
   while(I2C1STATbits.TRSTAT) {;}
   if(I2C1STATbits.ACKSTAT)
   { // if this is high, slave has not acknowledged
        return FALSE;
    }
    Else
   {
              return TRUE;
    }
}

// Receive a byte from the slave
BYTE i2c_master_recv(void)
{
    // Clear the previous data from temp. Rx buffer.
    memset(bRxTemp_Buf, 0, sizeof(bRxTemp_Buf));

    I2C1CONbits.RCEN = 1;                    // start receiving data
    while(!I2C1STATbits.RBF) { ; }   // wait to receive the data
    // Convert the DWORD to BYTE
    dword_to_buf( I2C1RCV, &bRxTemp_Buf[0]);
    return I2C1RCV;                        // read and return the data
}
```

4. I2C Master Lib API Real World Usages to Interface with EEPROM Memory

This section describe application of I2C master library APIs to interface with EEPROM memory via I2C communication bus from PIC32MZ MCU.

EEPROM stands for Electrically Erasable Programmable Read-Only Memory. It is a type of non-volatile memory that can be used to store little bits of data. Data will remain on EEPROM when power is disconnected, (like an external hardrive). The EEPROM chip has a life span in the tens or hundreds of thousands of read/writes. This code based on PIC32MZ (act as the I2C Master) and AT24CM02 EEPROM (act as the I2C slave) which communicating via I2C Bus.

The Microchip AT24CM02 is a 2Mb Serial EEPROM utilizing an I2C (2-wire) serial interface. The device is an organized as one block of 256K x 8 and is an optimized for use in consumer and industrial applications (such as a spectroscopy configuration and experiment data storage) where reliable and dependable nonvolatile memory storage is essential.

4.1. AT24CM02 EEPROM Device Configuration

The descriptions of AT24CM02 EEPROM pins are as below:

PIN1: No Connect
PIN2: No Connect
PIN3: Device Address Input
PIN4: Ground
PIN5: Serial Data (SDA) Line
PIN6: Serial Clock (SCL) Line
PIN7: Write Protect
PIN8: Device Power Supply

PIN3 Device Address Input (A2): The A2 pin is a device address input that is hard-wired (directly to GND or to VCC) for compatibility with other two-wire Serial EEPROM devices. Connected it to Ground for configure device address to “0x51”.

PIN4 Ground (GND): The ground reference for the power supply and it connected to the system ground.

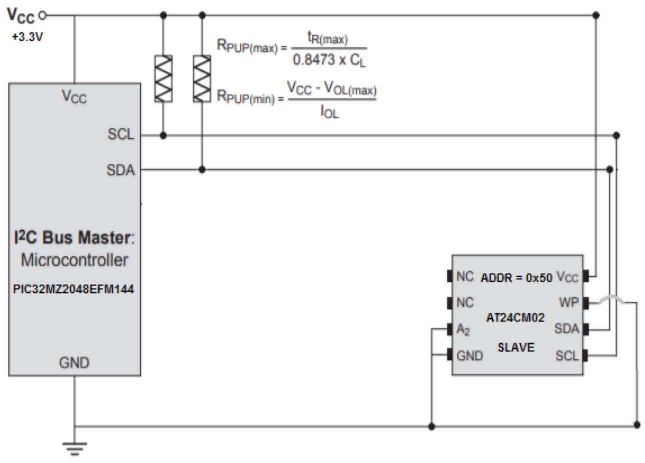
PIN5 SDA: The SDA pin is an open-drain bidirectional input/output pin used to serially transfer data to and from the device. The SDA pin must be pulled high using an external pull-up resistor (not to exceed 10 kΩ in value i.e. 3.3 kΩ) and may be wired with any number of other open-drain or open-collector pins from other devices on the same bus.

PIN6 SCL: It provides a clock to the device, which control the flow of data to and from the device. Command and input data present on the SDA pin is always latched in on the rising edge of SCL, while output data on the SDA pin is clocked out on the falling edge of SCL. The SCL pin must either be forced high when the serial bus is idle or pulled high using an external pull-up resistor.

PIN7 WP: The write-protect input is connected to GND, allows normal write operations.

PIN8 VCC: The VCC pin used to supply the source voltage to the device (+3.3 volt).

The PIC32MZ2048EFM144 MCU configuration using 2-wire serial EEPROM.


V_{CC}
+3.3V
$R_{PUP(max)} = \frac{t_{R(max)}}{0.8473 \times C_L}$
$R_{PUP(min)} = \frac{V_{CC} - V_{OL(max)}}{I_{OL}}$
V_{CC}
SCL
SDA
I²C Bus Master: Microcontroller
PIC32MZ2048EFM144
GND
NC
ADDR = 0x50
V_{CC}
NC
WP
AT24CM02
A_2
SDA
GND
SLAVE
SCL

4.2. Read and Write Cycle Process

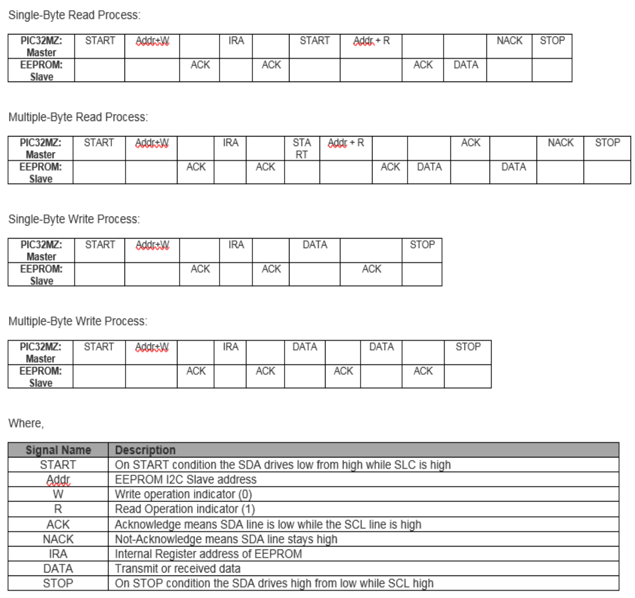
Read operations are initiated the same way as write operations with the exception that the Read/Write Select bit in the device address byte must be a logic ‘1’. There are three read operations:

- Current Address Read
- Random Address Read
- Sequential Read

All write operations for the AT24CM02 begin with the master sending a Start condition, followed by a device address byte with the R/W bit set to logic ‘0’, and then by the word address bytes. The data value(s) to be written to the device immediately follow the word address bytes. It support two write mode.

- Single Byte write mode.
- Whole page of 256 bytes write mode.

Refer the sequential flow of read and write operations:



Single-Byte Read Process:

PIC32MZ: Master	START	Addr+W		IRA		START	Addr + R			NACK	STOP
EEPROM: Slave			ACK		ACK			ACK	DATA		

Multiple-Byte Read Process:

PIC32MZ: Master	START	Addr+W		IRA		STA RT	Addr + R			ACK		NACK	STOP
EEPROM: Slave			ACK		ACK			ACK	DATA		DATA		

Single-Byte Write Process:

PIC32MZ: Master	START	Addr+W		IRA		DATA		STOP
EEPROM: Slave			ACK		ACK		ACK	

Multiple-Byte Write Process:

PIC32MZ: Master	START	Addr+W		IRA		DATA		DATA		STOP
EEPROM: Slave			ACK		ACK		ACK		ACK	

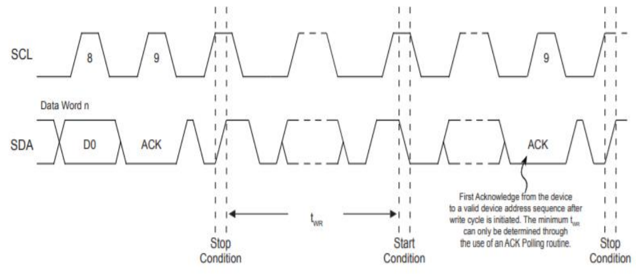
Where,

Signal Name	Description
START	On START condition the SDA drives low from high while SLC is high
Addr	EEPROM I2C Slave address
W	Write operation indicator (0)
R	Read Operation indicator (1)
ACK	Acknowledge means SDA line is low while the SCL line is high
NACK	Not-Acknowledge means SDA line stays high
IRA	Internal Register address of EEPROM
DATA	Transmit or received data
STOP	On STOP condition the SDA drives high from low while SCL high

4.3. Write Cycle Time

The length of the self-timed write cycle (tWR) defined as the amount of time from the Stop condition that begins the internal write cycle to the Start condition of the first device address byte sent to the AT24CM02 that it subsequently responds to with an ACK. During the internally self-timed write cycle, any attempts to read from or write to the memory array will not be processed.

The default Write Cycle time is 10 milliseconds for AT24CM02 EEPROM, this could be implemented using "delay_ms()" routine. (The Delay Millisecond routine API Corresponding citation: (Patel, 2019))


SCL
8
9
9
Data Word n
SDA
D0
ACK
ACK
t_{WR}
First Acknowledge from the device to a valid device address sequence after write cycle is initiated. The minimum t_{WR} can only be determined through the use of an ACK Polling routine.
Stop Condition
Start Condition
Stop Condition

4.4. Code for PIC32MZ and AT24CM02 EEPROM Communication over I2C Bus

This sample code design to read and validate the serial number from AT24CM02 EEPROM using I2C master library API. Also validate the serial number, it restore to default in case it is an invalid serial number. Similar to the serial number there are more data which stored into AT24CM02 EEPROM (such as a spectroscopy instrument configuration for performing an experiment, error log, event log, correction angle data points, manufacturing information, factory default instrument configuration etc.).

```
#include <string.h>
#include <stdio.h>
#include <GenericTypeDefs.h>
#include <p32mz2048efm144.h>
// Lib files.
#include "i2c_master_api.h"          // I2C Master API lib.
// The address of AT24CM02_EEPROM when the AD2 pin is
connected to ground
#define AT24CM02_EEPROM_ADDRESS  ( 0x50 )
#define AT24CM02_EEPROM_PAGE_SIZE   ( 0x40 )
#define I2C_READ_EEPROM                        ( 1 )
#define I2C_WRITE_EEPROM                       ( 0 )

#define EEPROM_ADDR_SN                ( 0x0A00 )
#define LENGTH_SN                  ( 25 )
// To start/restart i2c communication from master.
static bool StartTransfer( BOOL restart )
{
    // Send the Start (or Restart) signal
    if(restart)
    {
        i2c_master_restart();
    }
    else
    {
        i 2c_master_setup();
         i2c_master_start();
    }
    return TRUE;
}
// To STOP i2c communication.
static void StopTransfer( void )
{
    // Send the Stop signal
       i2c_master_stop();
}

// To send one byte data from i2c master to slave.
static BOOL TransmitOneByte(UINT8 data)
{
       BOOL bSendRet;
      bSendRet = i2c_master_send(data);

    return bSendRet;
}
// Check whether i2c master in idle condition or not.
void IsI2C_master_idle( void )
{
    i2c_master_idle();        // wait for completion
    return;
}
// To Read data from desired address of EEPROM AT24CM02
BOOL ReadI2C_AT24CM02_EEPROM(BYTE bAddr, BYTE
bLength, BYTE * bRxData, BOOL bRestart)
{
    int     Index;
    BOOL     bSuccess = TRUE;
    UINT8   RxSlaveAddress = 0;
    // Set the slave address, left shifted by 1,
    // and then a 1 in lsb, indicating read operation.
    RxSlaveAddress = ((bAddr << 1) | I2C_READ_EEPROM);
    // Start and send the address to switch initiate read transfer
    if(bSuccess)
    {
         // Start the transfer to read the device.
           if( !StartTransfer(FALSE) )
           {
                bSuccess = FALSE;
           }

       // Transmit the address with the READ bit set
       if (TransmitOneByte(RxSlaveAddress))
       {
                bSuccess = TRUE;
       }
       else
       {
          bSuccess = FALSE;
       }
    }

    // Read the data from the desired address
    if(bSuccess)
    {
           for (Index = 0; Index < bLength; Index++)
           {
                bRxData[Index] = i2c_master_recv(); // receive
another byte from the bus
          if(Index == (bLength - 1))
               i2c_master_ack(1);   // send NACK (1): master needs
no more bytes
          else
              i2c_master_ack(0); // send ACK (0): master wants
another byte!
           }
      }

    // End the transfer
      i2c_master_stop(); // send STOP: end transmission, give up bus
      return bSuccess;
}
BOOL WriteI2C_AT24CM02_EEPROM(BYTE bAddr, BYTE
bLength, BYTE * bTxData, BOOL bSendStop)
{
    UINT8    i2cData[258];
    int       Index;
    BOOL     bSuccess = TRUE;
    UINT8    TxSlaveAddress = 0;

    // Set the slave address, left shifted by 1,
    // which clears bit 0, indicating a write operation.
    TxSlaveAddress = ((bAddr << 1) | I2C_WRITE_EEPROM);

      //Load the device I2C address into the Tx buffer
    i2cData[0] = TxSlaveAddress;
      bLength++;
      for (Index = 1; Index < bLength; Index++)
      {//Copy the data into the Tx buffer
           i2cData[Index] = bTxData[Index - 1];
      }
    // Start the transfer to write data to the device
    if( !StartTransfer(FALSE) )
    {
      bSuccess = FALSE;
    }
//-------------------------------------------------------------------
// Transmit all data
    Index = 0;
    while( bSuccess && (Index < bLength) )
    {
       // Transmit a byte
       if (TransmitOneByte(i2cData[Index]))
       {
          // Transmission successful
          // Advance to the next byte
          Index++;
       }
       else
       {
          // Transmission was not successful
          bSuccess = FALSE;
          break;
       }
    }
    // End the transfer (hang here if an error occured)
      if (TRUE == bSendStop)
      {
      StopTransfer();
      }
      return bSuccess;
}

// To implementation of write cycle time for AT24CM02 EEPROM
BOOL WaitForEEPROMWrite_AT24CM02(BYTE bAddr)
{
    BOOL     bSuccess = TRUE;
      UINT8    TxSlaveAddress = 0;
TxSlaveAddress = ((bAddr << 1) | I2C_WRITE_EEPROM);

      while(1)
      {
                i2c_master_start();   // Send the Start Bit

                // Transmit just the EEPROM's address
                if (! TransmitOneByte(TxSlaveAddress))
                {       // Check the write error here.
                        bSuccess = FALSE;
```

```
                    if(!bSuccess)
                    {
                    bSuccess = FALSE;
                        break;
                    }
                }
                if(I2C1STATbits.ACKSTAT == 0) {
                  // EEPROM is back
                    i2c_master_stop();
                    break;
                }
                i2c_master_stop();
        }
        return bSuccess;
}
// To read data from EEPROM location.
static BOOL AT24CM02_EEPROMReadData(WORD wAddr, BYTE
bCount, BYTE * buf)
{
    BOOL bRet = TRUE;
    BYTE ee_str[66];

    ee_str[0] = (wAddr / 0x100) & 0xff;
    ee_str[1] = wAddr & 0xff;
    /* Check for I2C Master status before perform an any operation. */
    IsI2C_master_idle();
    bRet                                                &=
WriteI2C__AT24CM02_EEPROM(AT24CM02_EEPROM_ADDRES
S, 2, ee_str, TRUE);

    /* Check for I2C Master status before perform an any operation. */
    IsI2C_master_idle();
    bRet                                                 =
WaitForEEPROMWrite_AT24CM02(AT24CM02_EEPROM_ADDR
ESS);
    bRet                                                &=
ReadI2C_AT24CM02_EEPROM(AT24CM02_EEPROM_ADDRESS
, bCount, buf, FALSE);
    return bRet;
}
// To write Data to the EEPROM.
static BOOL AT24CM02_EEPROMWriteData(WORD wAddr, BYTE
bCount, const BYTE * buf)
{
    BOOL bRet = TRUE;
    BOOL bRetWR = TRUE;
    BYTE bLength;
    BYTE ee_str[66];
    while (bCount)
    {
        /* find # bytes at start of block so will end on page boundary */
        bLength = AT24CM02_EEPROM_PAGE_SIZE- (BYTE)
(wAddr % LG_EEPROM_PAGE_SIZE);
                bLength = bCount;

        /* prepare command for next address */
        ee_str[0] = (wAddr / 0x100) & 0xff;
        ee_str[1] = wAddr & 0xff;

        /* move data to I2C shift string */
        memcpy(&ee_str[2], buf, bLength);

            /* Write the data to the EEPROM and wait for it to
complete */
        bRetWR                                           =
WriteI2C__AT24CM02_EEPROM(AT24CM02_EEPROM_ADDRES
S, bLength + 2, ee_str, TRUE);
        bRet                                             =
WaitForEEPROMWrite_AT24CM02(AT24CM02_EEPROM_ADDR
ESS);
        /* decrement bytes remaining */
        bCount -= bLength;
        /* increment pointer into input buffer */
        buf += bLength;
        /* and increment EEPROM address */
        wAddr += (WORD) bLength;
    } /* loop over this block transfer */

/*********************************************************
**************
* >>>>> DEV NOTES <<<<<
* For the Write operation inverted the return status because with
* PIC32MZ I2C EEPROM returns TRUE on successfully write and
* failure return FALSE.
**********************************************************
*************/
    if ((!bRetWR))
        bRet = FALSE;    // FAILED to write the data.
    else
        bRet = TRUE;     // Successfully wrote the data.

    return bRet;
}
/* Read serial number from the EEPROM and if its invalid then restore
*   it to default serial number.
*/
void main()
{
    unsigned char value;
      WORD wLen = LENGTH_SN;
      BYTE * bpSerialNum;
      BYTE bSN[ LENGTH_SN ];
      BYTE          bSNStr[LENGTH_SN]          =          {
      'i','2','c','-','M','A','S','T','E','R',

      '-','A','T','2','4','C','M','0','2','-',
                                            'T','E','S','T','.'};

    // Set performance to ultra rad
    set_performance_mode();
    // Moved all the ANSEL, TRIS and LAT settings to their own
function
```

```
    setup_ports();
    // Enable multi-vectored interrupts mode
    INTCONbits.MVEC = 1;
    // No need to set up PPS, I2C hardware is fixed to certain pins.
SCL1 = RA14, SDA1 = RA15
    // Initialise I2C1 at 100kHz
    i2c_master_setup();
      /* Read the   serial number string from EEPROM */
      AT24CM02_EEPROMReadData(  (WORD)EEPROM_ADDR_
SN, &wLen, bpSerialNum );
      // Validate the serial number
      if ( ( bpSerialNum[0] < 0x20 ) || ( bpSerialNum[0] > 0x7F ) )
      {    /* if not printable serial number then                */
            /* load the default string, and update EEPROM */
            memcpy(&bSN[0], &bSNStr[0], LENGTH_SN);

      AT24CM02_EEPROMWriteData( (WORD)EEPROM_ADDR_
SN, LENGTH_SN, bSN );
      /* Allow AT24CM02 to complete an internal write cycle time of
10msec */
            delay_ms(10); // Corresponding citation : (Patel, 2019)
            /* Check for I2C Master status before perform an any
operation. */
            IsI2C_master_idle();
            /* Read the string from EEPROM */
      AT24CM02_EEPROMReadData(  (WORD)EEPROM_ADDR_
SN, *wLen, bpSerialNum );
      }
/* Wait 10ms before trying again so as not to overwhelm the
AT24CM02_EEPROM or the PIC32MZ's I2C peripheral */
    delay_ms(10); // Corresponding citation : (Patel, 2019)
}
```

5. Conclusions

These costume I2C master Library APIs can be useful for any PIC32 embedded system based project. There could be a numerous condition in real world applications where we need this I2C Master Library APIs to communicate with MCU, LCD driver, EEPROM memory, ASIC, voltage and temperature monitor chip via I2C communication bus from PIC32 MCU. These I2C Master Library APIs implemented successfully using PIC32MZ MCU hardware and MPLAB software stack that includes MPLAB X IDE, Tool chain and Harmony framework. The above example of storing and retrieving a serial number to AT24CM02 EEPROM from PIC32MZ demonstrated usages of via I2C master library APIs.

REFERENCES

[1] "Embedded Computing and Mechatronics with the PIC32 Microcontroller 1st Edition" by Kevin Lynch (Author), Nicholas Marchuk (Author), Matthew Elwin (Author).

[2] Data Sheet of "PIC32MZ Embedded Connectivity with Floating Point Unit (EF) Family " by Microchip.

[3] "PIC32MZ Milliseconds and Microseconds Delay Routines", by Abhishek N. Patel (Author), Electrical and Electronic Engineering, p-ISSN: 2162-9455 e-ISSN: 2162-8459, 2019; Vol 9(2): Page 41-44, Corresponding citation: (Patel, 2019).

[4] Data sheet of "AT24CM02 2-Mbit EEPROM" by ATMEL.

[5] I²C-bus Specification, Version 6.0, 4th of April 2014 by NXP.

[6] Application Note (AN735) for "Using the PICmicro® MSSP Module for Master I 2 CTM Communications" by Microchip.

[7] "Section 24. Inter-Integrated Circuit" by Microchip.

[8] AT24CM02 "https://www.microchip.com/wwwproducts/en/AT24CM02" by Microchip.

[9] PIC32MZEF family "https://www.microchip.com/design-centers/32-bit/pic-32-bit-mcus/pic32mz-ef-family" by Microchip.

[10] Microchip Developer Help "https://microchipdeveloper.com/harmony:start" by Microchip.