INTRODUCTION

The demand for cheap and portable nonvolatile memory devices remains steady as long-term memory storage is still integral to the development of many industrial and commercial technologies. For a lot of these markets, serial EEPROM devices are still seen as ideal, cost-effective solutions for nonvolatile memory embedded control applications. Despite the resurgence of other forms of nonvolatile memory, serial EEPROMs still make their case as a viable choice for applications and solutions that require portability, low-current and voltage operations, byte-per-byte operations and competitive price points. SPI and I²C synchronous serial protocols remain two of the most popular ways to interface with serial EEPROM devices. Catering to this, the Master Synchronous Serial Port (MSSP) module, built into most of the PIC® Microcontroller devices, provides a convenient platform for synchronous serial operations in both of these protocols.

This application note intends to demonstrate how to interface SPI and I²C serial EEPROM devices using MPLAB® X 3.10, the XC8 v1.34 compiler and the MPLAB® Code Configurator v2.25. The Explorer8 Development Board is used as the hardware development platform. The firmware written for this application note is built on the SPI and I²C function codes automatically generated by the MCC, providing references for byte read and write, buffer/page write, sequential read and write cycle polling operations. The code has been tested with EEPROMs of the MikroElectronika EEPROM and EEPROM2 Click™ Boards, and the SPI and I²C Plug-In Modules (PIMs) from Microchip’s Serial EEPROM PIM PICtail™ Pack.

SPi INTERFACE

The SPI protocol is best characterized by three features: it is synchronous, it designates a master device to communicate with slave device/s and it is a full-duplex system where data is exchanged between master and slave. SPI is a synchronous protocol that makes use of a clock signal to sync data transfer. No data transfer may occur unless a clock signal is present. The master device provides and controls this clock signal. All slave devices are controlled by this master clock and may not manipulate it. As data is being clocked out of either the master or slave/s, new data is being clocked in simultaneously. This is consistent with the full-duplex nature of the system. A Chip Select (CS) signal controls which particular slave device the master is communicating with, ensuring that only a single slave is engaged at one time.

For the examples and waveforms in this application note, a PIC16F1719 microcontroller is used as the master and a 2 Mbit serial bus EEPROM, mounted on the MikroElectronika EEPROM2 Click Boards, is the slave device.

STANDARD SPI SIGNALS

The SPI protocol makes use of four signal lines to arbitrate the flow of data in the communication system.

- **Chip Select (CS)** - This signal is used to select the slave device that the master will communicate with. Bringing the line to Active state will select the device.
- **Serial Clock (SCK)** - This is the clock signal generated by the master that controls when data is sent and read.
- **Serial Data Output (SDO)** - This is the signal line that carries the data sent out of the device.
- **Serial Data Input (SDI)** - This is the signal line that carries the data sent into the device.

**Note:** The SDO line of the master should be connected to the SDI line of the slave.
SPI MSSP INITIALIZATION VIA MPLAB® CODE CONFIGURATOR (MCC V2.25)

This section will guide the user in initializing the proper registers in order to implement SPI using the MSSP module available in most PIC devices. The MPLAB Code Configurator (MCC) is used to make this process easier and more intuitive. Upon opening a new project and launching the MCC plug-in, select the MSSP module in the Device Resources sidebar, marked in red in Figure 1. From the drop-down options, select SPI Master. This is marked in blue in Figure 1.

![Selecting MSSP Module in MCC](image)

In order to configure the MSSP module for the SPI operation in PIC devices, several registers need to be properly initialized. To match the configuration of the slave device, the SPI mode \((0,0)\) will be used, where the SCK signal Idles low, the data changes at the falling edge of the clock and is assumed valid at the rising edge.
SSPx Status Register (SSPxSTAT)

The SSPxSTAT register holds all of the status bits associated with the MSSP module. In SPI mode, the SMP bit determines the part of the input to be sampled. The CKE bit determines which edge of the clock the data is transmitted.

The BF bit indicates if the data byte transfer has already been completed. It is best to ensure the BF bit is cleared before starting any read or write operation. Figure 2 shows the position of the bits in the SSPxSTAT register and Figure 3 shows which parts of the configuration screen in MCC correspond to bits in the SSPxSTAT register.

The configuration shown in Figure 3 will yield the following lines of code in spi.c after clicking the ‘Generate Code’ button in MCC (see Example 1).

EXAMPLE 1: SSPxSTAT MCC GENERATED CODE

```c
// BF RCInprocess_TXcomplete; SMP Sample At Middle; CKE Active to Idle;
SSP1STAT = 0x40;
```
SSPx Control Register 1 (SSPxCON1)

The SSPxCON1 is one of the configuration registers for the MSSP module. It holds several indicator bits and the select bits to be configured in order to put the module in the desired mode. In SPI mode, the SSPEN bit needs to be set to enable the serial port. When enabled, the SCK, SDO and SDI pins are configured for SPI operation.

The CKP bit determines whether the clock Idles at a low or high level. The SSPM<3:0> bits determine the Synchronous Serial mode the module will operate in, as well as clock preferences. Figure 4 shows the position of the bits in the SSPxCON1 register, and Figure 5 shows which parts of the configuration screen in MCC correspond to bits in the SSPxCON1 register.

FIGURE 4: SSPxCON: SSPx CONTROL REGISTER 1 FOR SPI CONFIGURATION

FIGURE 5: CONFIGURING SSPxCON1 BITS ON MCC

The configuration shown in Figure 5 will yield the following lines of code in spi.c after clicking the 'Generate Code' button in MCC (see Example 2).

EXAMPLE 2: SSPxCON1 MCC GENERATED CODE

```c
// SSPEN enabled; WCOL no_collision; SSPOV no_overflow; CKP Idle:Low, Active:High; SSPM FOSC/4;
SSP1CON1 = 0x20;
```
For devices with Peripheral Pin Select (PPS) functionality, such as the PIC16F1719, the MCC would also automatically generate code to map the SDO, SDI and SCK pins to the pins selected in the pin manager, and initializes them as input or output accordingly. For other devices without PPS functionality, MCC configures the pins as input or output, as needed. Figure 6 shows the pins selected as SDO, SDI and SCK. Example 3 shows the code snippet generated to implement the selection via the Peripheral Pin Select (PPS) feature of this device. As for the CS pin, the user can select any unused I/O pin, and connect this to the slave device’s CS pin. The selected CS pin should start at the opposite level needed to activate the slave (i.e., if CS held low activates the slave, then it should start as high at Reset). This can be accomplished in the GPIO module in MCC by checking the appropriate box as shown in Figure 7. The pin can also be renamed for coding convenience.

**FIGURE 6: MSSP PIN ASSIGNMENTS**

**EXAMPLE 3: MSSP PIN ASSIGNMENT CODE**

```c
bool state = GIE;
GIE = 0;
PPSLOCK = 0x55;
PPSLOCK = 0xAA;
PPSLOCKbits.PPSLOCKED = 0x00;
// unlock PPS
// RC3->MSSP:SCK
SSPCLKPPSbits.SSPCLKPPS = 0x13;
// RC3->MSSP:SCK
RC3PPSbits.RC3PPS = 0x10;
// RC4->MSSP:SDI
SSPDATPPSbits.SSPDATPPS = 0x14;
// RC5->MSSP:SDO
RC5PPSbits.RC5PPS = 0x11;
PPSLOCK = 0x55;
PPSLOCK = 0xAA;
PPSLOCKbits.PPSLOCKED = 0x01;
// lock PPS
GIE = state;
```
FIGURE 7: SETTING THE CS PIN IN MCC

Write Enable
In order to begin interacting with the EEPROM, the CS line should be activated – brought low in the case for the EEPROM used here. This signals the slave to listen for the master’s SCK and SDO signals. The transitions of the CS line should bookend all transactions between the master and slave.

To begin writing to the EEPROM array or Status register, the WRITE ENABLE command must be sent by the master. The Write Enable (WEL) bit of the Status register is cleared by issuing a WRITE DISABLE command (WRDI) or if the device is powered down, or once a write cycle is completed. Figure 8 shows an example of the WRITE ENABLE command.

COMMON SPI SERIAL EEPROM OPERATIONS

TABLE 1: SAMPLE INSTRUCTION SET FOR THE SPI SERIAL BUS EEPROM

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
<th>Instruction Format/Opcode</th>
</tr>
</thead>
<tbody>
<tr>
<td>WREN</td>
<td>Write Enable</td>
<td>0000 0110</td>
</tr>
<tr>
<td>WRDI</td>
<td>Write Disable</td>
<td>0000 0100</td>
</tr>
<tr>
<td>RDSR</td>
<td>Read Status Register</td>
<td>0000 0101</td>
</tr>
<tr>
<td>WRSR</td>
<td>Write Status Register</td>
<td>0000 0001</td>
</tr>
<tr>
<td>READ</td>
<td>Read from Memory Array</td>
<td>0000 0011</td>
</tr>
<tr>
<td>WRITE</td>
<td>Write to Memory Array</td>
<td>0000 0010</td>
</tr>
</tbody>
</table>
For this EEPROM, the WRITE ENABLE command opcode is 0x06. Refer to the EEPROM’s data sheet for its specific command opcodes.

**Status Register Read**

The EEPROM’s Status register holds the bits that show the current condition of the EEPROM. The most important indicators for users to keep track of are the WEL (Write Enable) bit and the WIP (Write in Progress) bit. If the WEL bit is set, writing to the EEPROM’s data array is enabled. If the WIP bit is set, a write cycle is in progress. With this in mind, it is good programming practice to check these bits first before attempting write or read operations to avoid collisions. To read from the EEPROM Status register, bring the CS line low and send the EEPROM Read Status Register (RDSR) opcode (0x05 for this EEPROM). The Status register is then shifted out on the slave EEPROM’s SDO pin and into the Master’s SDI pin on the next succeeding clocks. Figure 9 and Figure 10 show the RDSR command being used to check if the WEL and WIP bits are set.
FIGURE 9: READ STATUS REGISTER COMMAND (WEL BIT SET)

FIGURE 10: READ STATUS REGISTER COMMAND (WEL+WIP BITS SET)
Byte and Buffer Write

Once the WEL bit is set and the CS line is brought low, the EEPROM write opcode needs to be sent (0x02 for this EEPROM), followed by the target starting address byte/s, with the Most Significant Byte (MSB) sent first. The data bytes are then clocked in last. Once the CS line is toggled at the end of this command, the internal write cycle is initiated.

The WIP bit of the Status register can now be polled to check when the write is finished (more on this later).

Multiple bytes can be written by continuously sending data bytes to the EEPROM device without toggling the CS line. However, users should be mindful of the page size of the device and starting address, so as not to overwrite previously stored data. Data exceeding the allotted page size will warp back to the starting address of the page, overwriting what may have been written there. Figure 11 and Figure 12 show how to write a single byte of data and multiple bytes of data in an array to the EEPROM.
Byte Read and Buffer Read

To read from the EEPROM, bring the CS line low and send the EEPROM read opcode (0x03 for this EEPROM), followed by the target starting address byte/s, with the Most Significant Bytes sent first. The code clocks out the data from the SDI line by sending dummy data, consisting of zeros, to the SDO line as SPI is a data exchange protocol. Once the CS line is toggled at the end of this command, the transfer is finished. A multibyte read can be accomplished by continuously sending dummy data bytes to the EEPROM device and thus, providing it with the clock cycles needed to send in the data without toggling the CS line. Figure 13 and Figure 14 show how to read a single byte of data and multiple bytes of data in an array from the EEPROM.

FIGURE 13: READ BYTE COMMAND

FIGURE 14: BUFFER READ COMMAND
I\textsuperscript{2}C INTERFACE

The I\textsuperscript{2}C protocol shares the first two features of SPI: it is synchronous and is designed as a master-slave protocol. Similar to the SPI, I\textsuperscript{2}C also uses a clock signal to facilitate data transfer. No data transfer may occur unless a clock signal is present. While the master device is still the one that provides the clock, slaves may manipulate the clock by holding it low to prevent further data transfer if it is still busy (clock stretching). This is in direct contrast with the SPI master that does not allow slaves to manipulate the clock signal. Moreover, the SPI protocol operates in Full-Duplex mode, allowing data to be sent simultaneously from both the master and slave. While the I\textsuperscript{2}C protocol operates in Half-Duplex mode, the master and slave are allowed to send data but not at the same time. This is implemented by an 'Acknowledge' system. Another difference is that the SPI protocol requires a Chip Select (CS) connection for each slave device. For I\textsuperscript{2}C, only two connections between the master and slaves are needed, regardless of the number of slaves, as the slave address is sent over the data line instead of using a CS connection. While the I\textsuperscript{2}C protocol uses fewer pins, the SPI protocol is usually faster.

For the examples and waveforms in this part of the application note, a PIC16F1719 microcontroller is used as the master and the 8 kbit Serial I\textsuperscript{2}C Bus EEPROM, mounted on the MikroElectronika EEPROM Click Boards, is used as the slave device.

STANDARD I\textsuperscript{2}C SIGNALS

The I\textsuperscript{2}C protocol makes use of only two signal lines to control the flow of data in the communication system.

- **Serial Clock Line (SCL)**
  - This is the clock signal generated by the master that controls when data is sent and received.
  - It can be held low by any slave device if it is too busy to accept or send data.

- **Serial Data Line (SDA)**
  - This is the signal line that carries the data sent in and out between master and slave devices.
I\textsuperscript{2}C MSSP INITIALIZATION VIA MPLAB\textsuperscript{\textregistered} CODE CONFIGURATOR (MCC)

This section will guide the user in initializing the proper registers in order to implement I\textsuperscript{2}C using the MSSP module available in most PIC devices using the MPLAB Code Configurator (MCC).

Upon opening a new project and launching the MCC plug-in, select the MSSP module in the Device Resources sidebar, marked in red in Figure 15. From the drop-down options, select 'I\textsuperscript{2}C Master Interrupt' as the microcontroller will be the master device, as mentioned in this section’s introduction. This is marked in blue in Figure 15.

FIGURE 15: SELECTING THE MSSP MODULE IN MCC

In order to configure the MSSP module for I\textsuperscript{2}C operation in the PIC devices, several registers need to be properly initialized.
SSPx Status Register (SSPxSTAT)

In I^2C Master mode, the R/W bit indicates if a transmit is in progress. The BF bit indicates if the data transfer has already been completed. Figure 16 shows the position of the bits in the SSPxSTAT register.

**FIGURE 16: SSPxSTAT: SSPX STATUS REGISTER FOR I^2C CONFIGURATION**

<table>
<thead>
<tr>
<th>SMP</th>
<th>CKE</th>
<th>D/Ù</th>
<th>P</th>
<th>S</th>
<th>R/W</th>
<th>UA</th>
<th>BF</th>
</tr>
</thead>
<tbody>
<tr>
<td>bit 7</td>
<td>bit 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The following lines of code initialize the SSPxSTAT register. The MCC automatically generates this code when the ‘Generate Code’ button is pressed (see Example 4).

**EXAMPLE 4: SSPxSTAT MCC GENERATED CODE**

```c
// BF RCinprocess_TXcomplete; UA dontupdate; P stopbit_notdetected; S startbit_notdetected;
R_nW write_noTX; D_nA lastbyte_address;
SSP1STAT = 0x00;
```
SSPx Control Register 1 (SSPxCON)

In I²C mode, the SSPEN bit needs to be set to enable the serial port and configures SCL and SDA as the source of the serial port pins.

The SSPM<3:0> bits determine the Synchronous Serial mode the module will operate in, as well as clock preferences and the number of bits used for slave addresses when the module is in Slave mode. Figure 17 shows the position of the bits in the SSPxCON1 register and Figure 18 shows which parts of the configuration screen in MCC correspond to bits in the SSPxCON1 register. In this example, the I²C Serial Bus EEPROM uses 7-bit addressing.

The configuration shown in Figure 18 will yield the following lines of code in i2c.c after clicking the ‘Generate Code’ button in MCC (see Example 5).

EXAMPLE 5: SSPxCON1 MCC GENERATED CODE

```c
// SSPEN enabled; WCOL no_collision; SSPOV no_overflow; CKP Idle:Low, Active:High; SSPM FOSC/4_SSPxADD;
SSP1CON1 = 0x28;
```
SSPx Control Register 3 (SSPxCON3)

This register mostly holds control bits for \( \text{i}^2\text{C} \) functions. Of concern here is the SDAHT bit, that controls how long the hold time of SDA will be after the falling edge of SCL.

**Figure 19** shows the position of the bits in the SSPxCON3 register and **Figure 20** shows which parts of the configuration screen in MCC correspond to bits in the SSPxCON3 register.

**FIGURE 19: SSPxCON3: SSPx CONTROL REGISTER 3 FOR \( \text{i}^2\text{C} \) CONFIGURATION**

<table>
<thead>
<tr>
<th>Bit</th>
<th>R-0/0</th>
<th>R/W-0/0</th>
<th>R-0/0</th>
<th>R/W-0/0</th>
<th>R-0/0</th>
<th>R/W-0/0</th>
<th>R-0/0</th>
<th>R/W-0/0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>ACKTIM(9)</td>
<td>PCIE</td>
<td>SCIE</td>
<td>BOEN</td>
<td>SDAHT</td>
<td>SBCDE</td>
<td>AHEN</td>
<td>DHEN</td>
</tr>
</tbody>
</table>

**FIGURE 20: CONFIGURING SSPxCON3 BITS ON MCC**

The configuration shown in **Figure 20** will yield the following lines of code in `i2c.c` after clicking the ‘Generate Code’ button in MCC (see **Example 6**).

**EXAMPLE 6: SSPxCON3 MCC GENERATED CODE**

```c
// BOEN disabled; AHEN disabled; SBCDE disabled; SDAHT 100ns; DHEN disabled; ACKTIM ackseq;
PCIE disabled; SCIE disabled;
SSP1CON3 = 0x00;
```
SSPx Address and Baud Rate Register (SSPxADD)

For I2C Master mode, this register holds the value of the Baud Rate clock divider (i.e., the SCL clock period). This is computed by the formula:

\[ \frac{(ADD_{<7:0>} + 1) \times 4}{F_{Osc}} \]

Figure 21 shows the position of the bits in the SSPxADD register. Figure 22 shows which parts of the configuration screen in MCC correspond to bits in the SSPxADD register.

The configuration shown in Figure 22 will yield the following lines of code in `i2c.c` after clicking the ‘Generate Code’ button in MCC (see Example 7).

```
// Baud Rate Generator Value: SSP1ADD 3;
SSP1ADD = 0x03;
```

Example 7: SSPxADD MCC GENERATED CODE
For devices with Peripheral Pin Select (PPS) functionality, such as the PIC16F1719, the MCC would also automatically generate code to map the SCL and SDA pins to the pins selected in the pin manager, and configure these pins as input or output accordingly. Figure 23 shows the pins selected as SCL and SDA. Example 8 shows the code snippet generated to implement the selection via the Peripheral Pin Select (PPS) feature of this device. Also note that the \( \text{I}^2\text{C} \) function driver, provided by the MCC, is interrupt-based, and so the user must enable global and peripheral interrupts for it to work. To enable global and peripheral interrupts, uncomment the lines implementing the `INTERRUPT_GlobalInterruptEnable()` and `INTERRUPT_PeripheralInterruptEnable()` functions on the MCC-generated `main.c` file, as shown in Figure 24.

**FIGURE 23:  \( \text{I}^2\text{C} \) MSSP PIN ASSIGNMENTS**

**EXAMPLE 8:  MSSP PIN ASSIGNMENT CODE**

```c
bool state = GIE;
GIE = 0;
PPSLOCK = 0x55;
PPSLOCK = 0xAA;
PPSLOCKbits.PPSLOCKED = 0x00; // unlock PPS
SSPCLKPPSbits.SSPCLKPPS = 0x13;  // RC3->MSSP:SCL
RC3PPSbits.RC3PPS = 0x10;
// RC3->MSSP:SCL
SSPDATPPSbits.SSPDATPPS = 0x14; // RC4->MSSP:SDA
RC4PPSbits.RC4PPS = 0x11;
// RC4->MSSP:SDA
PPSLOCK = 0x55;
PPSLOCK = 0xAA;
PPSLOCKbits.PPSLOCKED = 0x01; // lock PPS
GIE = state;
```
void main(void) {
    // initialize the device
    SYSTEM_Initialize();

    // When using interrupts, you need to set the Global and Peripheral Interrupt Enable bits
    // Use the following macros to:

    // Enable the Global Interrupts
    INTERRUPT_GlobalInterruptEnable();

    // Enable the Peripheral Interrupts
    INTERRUPT_PeripheralInterruptEnable();

    // Disable the Global Interrupts
    //INTERRUPT_GlobalInterruptDisable();

    // Disable the Peripheral Interrupts
    //INTERRUPT_PeripheralInterruptDisable();
}
COMMON I²C SERIAL EEPROM OPERATIONS

- Byte Write
- Multibyte Write
- Page Write
- Acknowledge Polling
- Write-Protect
- Address Read
- Sequential Read

Byte Write

The byte write operation in I²C can be broken down into the following elements: The Start condition, the I²C slave address byte, the EEPROM address byte, the data byte and the Stop condition. For this EEPROM, only a single byte of address data is used. Other EEPROMs might use multiple byte addresses.

START BIT AND I²C SLAVE ADDRESS BYTE TRANSMISSION

All I²C commands must begin with a Start condition. This consists of a high-to-low transition of the SDA line while the SCL is high. After the Start condition, the I²C slave address byte is sent, consisting of the device 7-bit I²C slave address (0xA for this EEPROM), and a Read/Write bit to identify the operation to be performed. For write operations, the R/W bit is pulled low.

SENDING THE EEPROM ADDRESS BYTE

After the I²C slave EEPROM has Acknowledged the receipt of the I²C address, the master should begin transmitting the EEPROM address byte. In the case that the EEPROM is using multiple byte addresses, the bytes should be sent in the order of decreasing significance. The EEPROM should respond with an ACK bit for every byte sent by the master.

SENDING THE DATA BYTE AND THE STOP BIT

Once the EEPROM address byte/s are sent and the ACK is received, the data byte can now be sent. The EEPROM should respond by sending an ACK bit after every byte sent by the master. After this, the master will generate the Stop condition, signifying that it does not have any more bytes to write. The Stop condition is achieved by creating a low-to-high transition of the SDA line while the SCL line is high.

Figure 25 shows the entire byte write procedure from Start-to-Stop conditions. In this case, the EEPROM only uses a single byte for addressing. 0x00 is used as the address where the data will be written and 0xA5 as the data byte to be transmitted.

FIGURE 25: BYTE WRITE COMMAND
Multibyte and Page Write

Writing multiple bytes starts off similar enough to the byte write operation. The Start bit, I²C slave address byte and EEPROM address byte/s are all sent and Acknowledged in that order. However, instead of sending a Stop condition after the first data byte has been transmitted and Acknowledged, the master just keeps on sending more data bytes consecutively. This EEPROM accepts up to 16 bytes to be written in a single write cycle. The page size of the EEPROM determines how many bytes of data can be consecutively written with the I²C slave address and EEPROM address bytes being sent only once. It is very important to point out that page write operations are limited to writing bytes within a single physical page, regardless of how many bytes are actually being written.

This means that if the number of bytes being written exceeds the page limit, the excess bytes will wrap back to the starting address of the page, overwriting data already written there. Once all bytes are sent, the master will initiate the Stop condition, thus beginning the internal write cycle.

Figure 26 shows the first three bytes sent in a buffer write operation. Note that immediately after the EEPROM Acknowledges the receipt of a byte, the master begins the transmission of the next byte.
Byte Read

The byte read operation in I^2C starts off similarly to the byte write as the EEPROM address should be written to the slave first, before the slave EEPROM can send the data from that address. The Start condition is sent first, then the I^2C slave address byte, then the EEPROM address byte/s. After sending the EEPROM address byte/s, the Stop condition is sent instead of data bytes. After the I^2C master initiates the read operation with a new Start condition, the I^2C slave address byte is sent with the LSB pulled high to signify a read operation. The I^2C master then sends nine clock pulses and the slave sends the single byte of data needed.

START BIT AND I^2C SLAVE ADDRESS BYTE TRANSMISSION

All I^2C commands must begin with a Start condition. This consists of a high-to-low transition of the SDA line while the SCL is high. After the Start condition, the I^2C slave address and direction byte are sent, consisting of the device 7-bit I^2C slave address (0xA for this EEPROM), and a Read/Write bit to determine the operation to be performed.

To initiate the byte read operation, the target EEPROM data address must be written to the EEPROM.

To achieve this, the R/W bit is set low to indicate that the I^2C master is writing to the slave. After each byte written by the master, the slave will hold the SDA line low, indicating the preceding byte was received.

SENDING THE ADDRESS BYTE

After the EEPROM has Acknowledged the receipt of the I^2C slave address byte, the master should begin transmitting the EEPROM address byte. In the case that the EEPROM is using multiple byte addresses, the bytes should be sent in the order of decreasing significance. The EEPROM should respond with an ACK bit after each byte is sent by the master.

SENDING THE STOP BIT

Once the address byte/s are sent and the ACK is received in read operations, the Stop bit is sent by the master at this point. The EEPROM should respond by sending an ACK bit.

RECEIVING THE DATA BYTE

A new Start condition is generated and the I^2C slave address byte is sent once again, but with the R/W bit set high to signify that the master wants to perform a read operation. Once the I^2C slave address byte is Acknowledged by the I^2C slave, the I^2C master will send nine clock pulses and the slave will respond by sending the data byte. After the master has received the data byte, it will release the SDA line at the ninth clock cycle. This is a Not Acknowledge (NACK) condition. This signals that the master is not requesting anymore data from the slave.

Figure 27 and Figure 28 show the entire byte read procedure. Figure 27 shows how the EEPROM address from where the data should be read is sent to the slave device. Figure 28 shows the read command and the data byte being sent from the slave. In this case, the EEPROM only uses a single byte for addressing. 0x00 is used as the EEPROM address where the data will be written and 0xA5 as the data byte to be transmitted.
FIGURE 27: SENDING THE ADDRESS FOR THE READ COMMAND

FIGURE 28: BYTE READ COMMAND
Multibyte/Sequential Read

Unlike page write operations that are limited by the physical page size of the device, a sequential read can read the entire contents of the memory in a single operation. Reading multiple bytes starts off similar enough to the byte read operation. The Start bits, \(^2\text{C}\) slave address byte, EEPROM address byte/s and Stop condition are all sent and Acknowledged in that order. A new Start condition is generated and the \(^2\text{C}\) slave address byte, with the LSB set high, should be sent.

However, instead of the \(^2\text{C}\) master sending a NACK bit after the first byte has been transmitted, the \(^2\text{C}\) master pulls the line low, sending an ACK bit, signifying that there is more data requested by the master. The master sends an ACK bit after each byte it receives, except after the last byte, where it will send the NACK bit, indicating that the master is not requesting any more data to be sent. Once all bytes are received, the master will initiate the Stop condition to end the operation.

Figure 29 shows a sequential read operation. Note that each byte successfully sent is followed by an ACK bit, save for the last byte, which is followed by a NACK bit.

FIGURE 29: SEQUENTIAL READ OPERATION
Acknowledge Polling

While most EEPROM data sheets specify a write cycle time, some write cycles might be shorter than this period. Specifying a delay period therefore, might not be efficient. Hence, it is recommended that users use Acknowledge polling to check if the current write cycle is finished.

This is done by continuously sending a Start condition and the I²C slave address byte, with the LSB set low (signifying a write operation), until an Acknowledge bit is detected. This is because when a write cycle is in progress, EEPROMs will not Acknowledge commands. Figure 30 shows an Acknowledge polling operation before an address write operation.

FIGURE 30: ACKNOWLEDGE POLLING
CONCLUSION

This application note illustrates the ease and efficiency with which interfacing serial EEPROM devices is accomplished by using the MSSP modules and the MCC. Basic operations in the SPI and I²C protocols were discussed and shown step-by-step. The code is highly portable and can be used on most 8-bit PIC microcontrollers and serial EEPROM devices, with just minor modifications. Using the code provided, users can begin to build their own SPI and I²C EEPROM applications. If a more complex, or a more deconstructed firmware design is needed, users can always fall back on the MCC-generated SPI and I²C functions that formed the backbone of the provided driver files. This document has demonstrated that when paired with the MSSP module and the MPLAB® Code Configurator, building solutions involving serial EEPROMs that use SPI or I²C does not have to be as tedious or as labor-intensive as it used to be.
APPENDIX A: CONNECTING THE EEPROMs TO THE MICROCONTROLLER

FIGURE A-1: SPI CONNECTION DIAGRAM
FIGURE A-2: \( \text{I}^2\text{C} \) CONNECTION DIAGRAM
APPENDIX B: SOURCE CLICK™ BOARD CONNECTION DIAGRAMS

FIGURE B-1: EEPROM CLICK™ BOARD CONNECTION DIAGRAM

FIGURE B-2: EEPROM2 CLICK™ BOARD CONNECTION DIAGRAM
APPENDIX C: SOURCE CODE LISTINGS

Software License Agreement

The software supplied herewith by Microchip Technology Incorporated (the “Company”) is intended and supplied to you, the Company’s customer, for use solely and exclusively with products manufactured by the Company.

The software is owned by the Company and/or its supplier, and is protected under applicable copyright laws. All rights are reserved. Any use in violation of the foregoing restrictions may subject the user to criminal sanctions under applicable laws, as well as to civil liability for the breach of the terms and conditions of this license.

THIS SOFTWARE IS PROVIDED IN AN "AS IS" CONDITION. NO WARRANTIES, WHETHER EXPRESS, IMPLIED OR STATUTORY, INCLUDING, BUT NOT LIMITED TO, IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE APPLY TO THIS SOFTWARE. THE COMPANY SHALL NOT, IN ANY CIRCUMSTANCES, BE LIABLE FOR SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES, FOR ANY REASON WHATSOEVER.

EXAMPLE C-1: eeprom_spi.c

```
/*
EEPROM SPI Source File

Company:
    Microchip Technology Inc.

File Name:
eeprom_spi.c

Summary:
    This is the source file containing the EEPROM SPI functions.

Description:
    This header file provides implementations for driver APIs for all modules selected in the GUI.

Generation Information:
    Product Revision: MPLAB® Code Configurator - v2.25.2
    Device: PIC16F1719
    Driver Version: 2.00
    The generated drivers are tested against the following:
    Compiler: XC8 v1.34
    MPLAB: MPLAB X v2.35 or v3.00
*/

/*
Copyright (c) 2013 - 2015 released Microchip Technology Inc. All rights reserved.

Microchip licenses to you the right to use, modify, copy and distribute
Software only when embedded on a Microchip microcontroller or digital signal
controller that is integrated into your product or third party product
(pursuant to the sublicense terms in the accompanying license agreement).

You should refer to the license agreement accompanying this Software for additional information regarding your rights and obligations.

SOFTWARE AND DOCUMENTATION ARE PROVIDED "AS IS" WITHOUT WARRANTY OF ANY KIND,
EITHER EXPRESS OR IMPLIED, INCLUDING WITHOUT LIMITATION, ANY WARRANTY OF
MERCHANTABILITY, TITLE, NON-INFRINGEMENT AND FITNESS FOR A PARTICULAR PURPOSE.
IN NO EVENT SHALL MICROCHIP OR ITS LICENSORS BE LIABLE OR OBLIGATED UNDER
CONTRACT, NEGLIGENCE, STRICT LIABILITY, CONTRIBUTION, BREACH OF WARRANTY, OR
OTHER LEGAL EQUITABLE THEORY ANY DIRECT OR INDIRECT DAMAGES OR EXPENSES
INCLUDING BUT NOT LIMITED TO ANY INCIDENTAL, SPECIAL, INDIRECT, PUNITIVE OR
CONSEQUENTIAL DAMAGES, LOST PROFITS OR LOST DATA, COST OF PROCUREMENT OF
SUBSTITUTE GOODS, TECHNOLOGY, SERVICES, OR ANY CLAIMS BY THIRD PARTIES
(INCLUDING BUT NOT LIMITED TO ANY DEFENSE THEREOF), OR OTHER SIMILAR COSTS.
```
EXAMPLE C-1:  eeprom_spi.c (CONTINUED)

```c
/*
 * #include "mcc_generated_files/mcc.h"
 * #include "eeprom_spi.h"

void SPI_ByteWrite (uint8_t *addressBuffer, uint8_t addlen, uint8_t byteData) {
    uint8_t check;
    //Toggle CS line to start operation
    CS_LAT = 0;
    //Send Write Enable command
    SPI_Exchange8bit(EEPROM_WREN);
    //Toggle CS line to end operation
    CS_LAT = 1;
    //Check if WEL bit is set
    while(check != 2)
        check = SPI_ReadStatusRegister();
    //Toggle CS line to start operation
    CS_LAT = 0;
    //Send Write Command
    SPI_Exchange8bit(EEPROM_WRITE_EN);
    //Send address byte/s
    SPI_Exchange8bitBuffer(addressBuffer, addlen, NULL);
    //Send data byte
    SPI_Exchange8bit(byteData);
    //Toggle CS line to end operation
    CS_LAT = 1;
}

uint8_t SPI_ByteRead (uint8_t *addressBuffer, uint8_t addlen) {
    uint8_t readByte;
    //Toggle CS line to start operation
    CS_LAT = 0;
    //Send Read Command
    SPI_Exchange8bit(EEPROM_READ_EN);
    //Send address bytes
    SPI_Exchange8bitBuffer(addressBuffer, addlen, NULL);
    //Send Dummy data to clock out data byte from slave
    readByte = SPI_Exchange8bit(DUMMY_DATA);
    //Toggle CS line to end operation
    CS_LAT = 1;
    //return data byte read
    return(readByte);
}
```
EXAMPLE C-1:  
eeprom_spi.c (CONTINUED)

```c
uint8_t SPI_ReadStatusRegister(void)
{
    uint8_t statusByte;

    //Toggle CS line to start operation
    CS_LAT = 0;

    //Send Read Status Register Operation
    SPI_Exchange8bit(EEPROM_RDSR);
    //Send Dummy data to clock out data byte from slave
    statusByte = SPI_Exchange8bit(DUMMY_DATA);

    //Toggle CS line to end operation
    CS_LAT = 1;

    //return data byte read
    return(statusByte);
}

uint8_t SPI_WritePoll(void)
{
    uint8_t pollByte;

    //Read the Status Register
    pollByte = SPI_ReadStatusRegister();

    //Check if WEL and WIP bits are still set
    while(pollByte == 3)
    {
        pollByte = SPI_ReadStatusRegister();
    }

    //return 1 if WEL and WIP bits are cleared and the write cycle is finished
    return(1);
}

void SPI_SequentialWrite(uint8_t *addressBuffer, uint8_t addlen, uint8_t *writeBuffer, uint8_t buflen)
{
    //Toggle CS line to begin operation
    CS_LAT = 0;

    //Send Write Enable Command
    SPI_Exchange8bit(EEPROM_WREN);

    //Toggle CS line to end operation
    CS_LAT = 1;

    //Toggle CS line to start operation
    CS_LAT = 0;

    //Send Write Command
    SPI_Exchange8bit(EEPROM_WRITE_EN);
    //Send address bytes
    SPI_Exchange8bitBuffer(addressBuffer, addlen, NULL);
    //Send data bytes to be written
    SPI_Exchange8bitBuffer(writeBuffer, buflen, NULL);
}
```
EXAMPLE C-1:  eeprom_spi.c (CONTINUED)

```c
uint8_t SPI_SequentialRead(uint8_t *addressBuffer,uint8_t addlen, uint8_t *readBuffer, uint8_t buflen)
{
    //Toggle CS line to begin operation
    CS_LAT = 0;
    //Send Read Command
    SPI_Exchange8bit(EEPROM_READ_EN);
    //Send Address bytes
    SPI_Exchange8bitBuffer(addressBuffer,addlen,NULL);
    //Send dummy/NULL data to clock out data bytes from slave
    SPI_Exchange8bitBuffer(NULL,buflen,readBuffer);
    //Toggle CS line to end operation
    CS_LAT = 1;
}
```

EXAMPLE C-2:  eeprom_spi.h

```c
/**
 * EEPROM SPI Header File
 *
 * Company:  
 * Microchip Technology Inc.
 *
 * File Name:  
 * eeprom_spi.h
 *
 * Summary:  
 * This is the header file containing the EEPROM I2C functions.
 *
 * Description:  
 * This header file provides implementations for driver APIs for all modules selected in the GUI.
 *
 * Generation Information:  
 * Product Revision :  MPLAB® Code Configurator - v2.25.2
 * Device            :  PIC16F1719
 * Driver Version    :  2.00
 * The generated drivers are tested against the following:
 * Compiler          :  XC8 v1.34
 * MPLAB             :  MPLAB X v2.35 or v3.00
 */

/*
Copyright (c) 2013 - 2015 released Microchip Technology Inc.  All rights reserved.

Microchip licenses to you the right to use, modify, copy and distribute
Software only when embedded on a Microchip microcontroller or digital signal
controller that is integrated into your product or third party product
(pursuant to the sublicense terms in the accompanying license agreement).
*/
EXAMPLE C-2:  eeprom_spi.h (CONTINUED)

You should refer to the license agreement accompanying this Software for additional information regarding your rights and obligations.

SOFTWARE AND DOCUMENTATION ARE PROVIDED "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING WITHOUT LIMITATION, ANY WARRANTY OF MERCHANTABILITY, TITLE, NON-INFRINGEMENT AND FITNESS FOR A PARTICULAR PURPOSE. IN NO EVENT SHALL MICROCHIP OR ITS LICENSORS BE LIABLE OR OBLIGATED UNDER CONTRACT, NEGLIGENCE, STRICT LIABILITY, CONTRIBUTION, BREACH OF WARRANTY, OR OTHER LEGAL EQUITABLE THEORY ANY DIRECT OR INDIRECT DAMAGES OR EXPENSES INCLUDING BUT NOT LIMITED TO ANY INCIDENTAL, SPECIAL, INDIRECT, PUNITIVE OR CONSEQUENTIAL DAMAGES, LOST PROFITS OR LOST DATA, COST OF PROCUREMENT OF SUBSTITUTE GOODS, TECHNOLOGY, SERVICES, OR ANY CLAIMS BY THIRD PARTIES (INCLUDING BUT NOT LIMITED TO ANY DEFENSE THEREOF), OR OTHER SIMILAR COSTS.

# include "mcc_generated_files/spi.h"

#ifndef EEPROM_SPI_H
#define EEPROM_SPI_H

#ifdef __cplusplus
extern "C" {
#endif

#define EEPROM_READ_EN 0x03 // read data from memory
#define EEPROM_WREN 0x06 // set the write enable latch
#define EEPROM_WRITE_EN 0x02 // write data to memory array
#define EEPROM_RDSR 0x05 // read STATUS register

void SPI_ByteWrite (uint8_t *addressBuffer, uint8_t addlen, uint8_t byteData);
uint8_t SPI_ByteRead (uint8_t *addressBuffer,uint8_t addlen);
uint8_t SPI_ReadStatusRegister(void);
uint8_t SPI_WritePoll(void);
void SPI_SequentialWrite(uint8_t *addressBuffer, uint8_t addlen, uint8_t *writeBuffer,uint8_t buflen);
void SPI_SequentialRead(uint8_t *addressBuffer,uint8_t addlen, uint8_t *readBuffer, uint8_t buflen);

#ifdef __cplusplus
}
#endif
#endif /* EEPROM_SPI_H */
EXAMPLE C-3: SAMPLE MAIN FILE CALLING SPI FUNCTIONS

/**
 * Generated Main Source File
 *
 * Company:
 * Microchip Technology Inc.
 *
 * File Name:
 * main.c
 *
 * Summary:
 * This is the main file generated using MPLAB® Code Configurator
 *
 * Description:
 * This header file provides implementations for driver APIs for all modules selected in the GUI.
 * Generation Information :
 * Product Revision : MPLAB® Code Configurator - v2.25.2
 * Device : PIC16F1719
 * Driver Version : 2.00
 * The generated drivers are tested against the following:
 * Compiler : XC8 v1.34
 * MPLAB X : MPLAB X v2.35 or v3.00
 *
 * Copyright (c) 2013 - 2015 released Microchip Technology Inc. All rights reserved.
 *
 * Microchip licenses to you the right to use, modify, copy and distribute
 * Software only when embedded on a Microchip microcontroller or digital signal
 * controller that is integrated into your product or third party product
 * (pursuant to the sublicense terms in the accompanying license agreement).
 *
 * You should refer to the license agreement accompanying this Software for
 * additional information regarding your rights and obligations.
 *
 * SOFTWARE AND DOCUMENTATION ARE PROVIDED "AS IS" WITHOUT WARRANTY OF ANY KIND,
 * EITHER EXPRESS OR IMPLIED, INCLUDING WITHOUT LIMITATION, ANY WARRANTY OF
 * MERCHANTABILITY, TITLE, NON-INFRINGEMENT AND FITNESS FOR A PARTICULAR PURPOSE.
 * IN NO EVENT SHALL MICROCHIP OR ITS LICENSORS BE LIABLE OR OBLIGATED UNDER
 * CONTRACT, NEGLIGENCE, STRICT LIABILITY, CONTRIBUTION, BREACH OF WARRANTY, OR
 * OTHER LEGAL EQUITABLE THEORY ANY DIRECT OR INDIRECT DAMAGES OR EXPENSES
 * INCLUDING BUT NOT LIMITED TO ANY INCIDENTAL, SPECIAL, INDIRECT, PUNITIVE OR
 * CONSEQUENTIAL DAMAGES, LOST PROFITS OR LOST DATA, COST OF PROCUREMENT OF
 * SUBSTITUTE GOODS, TECHNOLOGY, SERVICES, OR ANY CLAIMS BY THIRD PARTIES
 * (INCLUDING BUT NOT LIMITED TO ANY DEFENSE THEREOF), OR OTHER SIMILAR COSTS.
 *
 * #include "mcc_generated_files/mcc.h"
 * #include "eeprom_spi.h"
 *
 * Main application
 *
 */
 void main(void) {
    // initialize the device
    SYSTEM_Initialize();
EXAMPLE C-3: SAMPLE MAIN FILE CALLING SPI FUNCTIONS (CONTINUED)

// When using interrupts, you need to set the Global and Peripheral Interrupt Enable bits
// Use the following macros to:

// Enable the Global Interrupts
// INTERRUPT_GlobalInterruptEnable();

// Enable the Peripheral Interrupts
// INTERRUPT_PeripheralInterruptEnable();

// Disable the Global Interrupts
// INTERRUPT_GlobalInterruptDisable();

// Disable the Peripheral Interrupts
// INTERRUPT_PeripheralInterruptDisable();

uint8_t writeBuffer[] = {0x1A, 0x2A, 0x4A, 0x8A};
uint8_t readBuffer[10];
uint8_t addressBuffer[] = {0xAB,0x00,0x10}; // Store the address you want to access here
uint8_t readByte;

// Writes one byte to the address specified
SPI_ByteWrite(&addressBuffer,sizeof(addressBuffer),0xA5);

// Wait for write cycle to complete
SPI_WritePoll();

// Reads one byte of data from the address specified
readByte = SPI_ByteRead(&addressBuffer,sizeof(addressBuffer));

// Intermission
__delay_ms(10);

// Writes the data in writeBuffer beginning from the address specified
SPI_SequentialWrite(&addressBuffer,sizeof(addressBuffer),&writeBuffer,sizeof(writeBuffer));

// Wait for write cycle to complete
SPI_WritePoll();

// Reads specified number of data bytes into the readBuffer array beginning from the address indicated
SPI_SequentialRead(&addressBuffer,sizeof(addressBuffer),&readBuffer,4);

// Stop here
while (1) {
    // Add your application code
}

***
End of File
*/
EXAMPLE C-4:  eeprom_i2c.c

/**
 * EEPROM I2C Source File
 *
 * Company:
 * Microchip Technology Inc.
 *
 * File Name:
 * eeprom_i2c.c
 *
 * Summary:
 * This is the source file containing the EEPROM I2C functions and constants.
 *
 * Description:
 * This header file provides implementations for driver APIs for all modules selected in the GUI.
 *
 * Generation Information :
 * Product Revision : MPLAB® Code Configurator - v2.25.2
 * Device : PIC16F1719
 * Driver Version : 2.00
 * The generated drivers are tested against the following:
 * Compiler : XC8 v1.34
 * MPLAB : MPLAB X v2.35 or v3.00
 *
 */

/*
Copyright (c) 2013 - 2015 released Microchip Technology Inc. All rights reserved.

Microchip licenses to you the right to use, modify, copy and distribute Software only when embedded on a Microchip microcontroller or digital signal controller that is integrated into your product or third party product (pursuant to the sublicense terms in the accompanying license agreement).

You should refer to the license agreement accompanying this Software for additional information regarding your rights and obligations.

SOFTWARE AND DOCUMENTATION ARE PROVIDED "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING WITHOUT LIMITATION, ANY WARRANTY OF MERCHANTABILITY, TITLE, NON-INFRINGEMENT AND FITNESS FOR A PARTICULAR PURPOSE. IN NO EVENT SHALL MICROCHIP OR ITS LICENSORS BE LIABLE OR OBLIGATED UNDER CONTRACT, NEGLIGENCE, STRICT LIABILITY, CONTRIBUTION, BREACH OF WARRANTY, OR OTHER LEGAL EQUITABLE THEORY ANY DIRECT OR INDIRECT DAMAGES OR EXPENSES INCLUDING BUT NOT LIMITED TO ANY INCIDENTAL, SPECIAL, INDIRECT, PUNITIVE OR CONSEQUENTIAL DAMAGES, LOST PROFITS OR LOST DATA, COST OF PROCUREMENT OF SUBSTITUTE GOODS, TECHNOLOGY, SERVICES, OR ANY CLAIMS BY THIRD PARTIES (INCLUDING BUT NOT LIMITED TO ANY DEFENSE THEREOF), OR OTHER SIMILAR COSTS.
*/

#include "mcc_generated_files/mcc.h"
#include "eeprom_i2c.h"

uint8_t timeOut = 0;

int I2C_ByteWrite(uint8_t *dataAddress, uint8_t dataByte, uint8_t addlen)
{
    uint8_t writeBuffer[PAGE_LIMIT+3];
    uint8_t buflen;

    //Copy address bytes to the write buffer so it can be sent first
    for(int i = 0; i < addlen; i++)
    
    return writeBuffer;
}
EXAMPLE C-4:  

eesrom_i2c.c (CONTINUED)

```c
{  
    writeBuffer[i] = dataAddress[i];
}  

//Check if this is an address write or a data write.
if(dataByte != NULL)
{
    writeBuffer[addlen] = dataByte;
    buflen = addlen+1;
}
else
    buflen = addlen;

//set status to Message Pending to send the data
I2C_MESSAGE_STATUS status = I2C_MESSAGE_PENDING;

//This variable is the built in acknowledge polling mechanism. This counts how many retries the system has already done to send the data.
timeOut = 0;

//While the message has not failed...
while(status != I2C_MESSAGE_FAIL)
{
    // Initiate a write to EEPROM
    I2C_MasterWrite(writeBuffer,buflen,SLAVE_ADDRESS,&status);

    // wait for the message to be sent or status has changed.
    while(status == I2C_MESSAGE_PENDING);
    // if transfer is complete, break the loop
    if (status == I2C_MESSAGE_COMPLETE)
        break;
    // if transfer fails, break the loop
    if (status == I2C_MESSAGE_FAIL)
        break;
    //Max retry is set for max Ack polling. If the Acknowledge bit is not set, this will just loop again until the write command is acknowledged
    if (timeOut == MAX_RETRY)
        break;
    else
        timeOut++;
}

// if the transfer failed, stop at this point
if (status == I2C_MESSAGE_FAIL)
    return 1;
}
```

```c
uint8_t I2C_ByteRead(uint8_t *dataAddress,uint8_t dataByte, uint8_t addlen)
{
    int check;

    //Write the address to the slave
    check = I2C_ByteWrite(dataAddress,NULL,addlen);

    //If not successful, return to function
    if(check == 1)
        return;
```
EXAMPLE C-4:  eeprom_i2c.c (CONTINUED)

```c
//Get ready to send data
I2C_MESSAGE_STATUS status = I2C_MESSAGE_PENDING;
//Set up for ACK polling
timeOut = 0;

//While the code has not detected message failure..
while(status != I2C_MESSAGE_FAIL)
{
    // Initiate a Read to EEPROM
    I2C_MasterRead(dataByte,1,SLAVE_ADDRESS,&status);

    // wait for the message to be sent or status has changed.
    while(status == I2C_MESSAGE_PENDING);

    // if transfer is complete, break the loop
    if (status == I2C_MESSAGE_COMPLETE)
        break;

    // if transfer fails, break the loop
    if (status == I2C_MESSAGE_FAIL)
        break;

    // check for max retry and skip this byte
    if (timeOut >= MAX_RETRY)
        break;
    else
        timeOut++;
}
}

int I2C_BufferWrite(uint8_t *dataAddress, uint8_t *dataBuffer, uint8_t addlen, uint8_t buflen)
{
    uint8_t writeBuffer[PAGE_LIMIT+3];
    I2C_MESSAGE_STATUS status = I2C_MESSAGE_PENDING;

    //Set Address as the bytes to be written first
    for(int i = 0; i < addlen; i++)
    {
        writeBuffer[i] = dataAddress[i];
    }

    //Ensure that the page limit is not breached so as to avoid overwriting other data
    if(buflen > PAGE_LIMIT)
        buflen = PAGE_LIMIT;

    //Copy data bytes to write buffer
    for(int j = 0; j < buflen; j++)
    {
        writeBuffer[addlen+j] = dataBuffer[j];
    }

    //Set up for ACK polling
    timeOut = 0;
    while(status != I2C_MESSAGE_FAIL)
    {
        // Initiate a write to EEPROM
        I2C_MasterWrite(writeBuffer,buflen+addlen,SLAVE_ADDRESS,&status);
    }
}
```
EXAMPLE C-4:  eeprom_i2c.c (CONTINUED)

// wait for the message to be sent or status has changed.
   while(status == I2C_MESSAGEPending);
// if transfer is complete, break the loop
   if (status == I2C_MESSAGE_COMPLETE)
      break;
   // if transfer fails, break the loop
   if (status == I2C_MESSAGE_FAIL)
      break;

   //check for max retry and skip this byte
   if (timeOut == MAX_RETRY)
      break;
   else
      timeOut++;
   }
   // if the transfer failed, stop at this point
   if (status == I2C_MESSAGE_FAIL)
      return 1;

void I2C_BufferRead(uint8_t *dataAddress, uint8_t *dataBuffer, uint8_t addlen,uint8_t buflen)
{
   int check = 0;
   I2C_MESSAGE_STATUS status = I2C_MESSAGE_PENDING;

   //Write Address from where to read
   check = I2C_ByteWrite(dataAddress,NULL,addlen);

   //check if address write is successful
   if(check == 1)
      return;

   //Set up for ACK polling
   timeOut = 0;

   while(status != I2C_MESSAGE_FAIL){
      // Initiate a Read to EEPROM
      I2C_MasterRead(dataBuffer,buflen,SLAVE_ADDRESS,&status);

      // wait for the message to be sent or status has changed.
      while(status == I2C_MESSAGE_PENDING);

      // if transfer is complete, break the loop
      if (status == I2C_MESSAGE_COMPLETE)
         break;

      // if transfer fails, break the loop
      if (status == I2C_MESSAGE_FAIL)
         break;

      // check for max retry and skip this byte
      if (timeOut == MAX_RETRY)
         break;
      else
         timeOut++;
   }
EXAMPLE C-5:  eeprom_i2c.h

/**
 * EEPROM I2C Header File
 *
 * Company:
 * Microchip Technology Inc.
 *
 * File Name:
 * eeprom_i2c.h
 *
 * Summary:
 * This is the header file containing the EEPROM I2C functions and constants.
 *
 * Description:
 * This header file provides implementations for driver APIs for all modules selected in the GUI.
 *
 * Generation Information:
 * Product Revision : MPLAB® Code Configurator - v2.25.2
 * Device : PIC16F1719
 * Driver Version : 2.00
 * The generated drivers are tested against the following:
 * Compiler : XC8 v1.34
 * MPLAB : MPLAB X v2.35 or v3.00
 */

/*
Copyright (c) 2013 - 2015 released Microchip Technology Inc. All rights reserved.

Microchip licenses to you the right to use, modify, copy and distribute Software only when embedded on a Microchip microcontroller or digital signal controller that is integrated into your product or third party product (pursuant to the sublicense terms in the accompanying license agreement).

You should refer to the license agreement accompanying this Software for additional information regarding your rights and obligations.

SOFTWARE AND DOCUMENTATION ARE PROVIDED "AS IS" WITHOUT WARRANTY OF ANY KIND,
EITHER EXPRESS OR IMPLIED, INCLUDING WITHOUT LIMITATION, ANY WARRANTY OF MERCHANTABILITY, TITLE, NON-INFRINGEMENT AND FITNESS FOR A PARTICULAR PURPOSE.
IN NO EVENT SHALL MICROCHIP OR ITS LICENSORS BE LIABLE OR OBLIGATED UNDER
CONTRACT, NEGLIGENCE, STRICT LIABILITY, CONTRIBUTION, BREACH OF WARRANTY, OR
OTHER LEGAL EQUITABLE THEORY ANY DIRECT OR INDIRECT DAMAGES OR EXPENSES
INCLUDING BUT NOT LIMITED TO ANY INCIDENTAL, SPECIAL, INDIRECT, PUNITIVE OR
CONSEQUENTIAL DAMAGES, LOST PROFITS OR LOST DATA, COST OF PROCUREMENT OF
SUBSTITUTE GOODS, TECHNOLOGY, SERVICES, OR ANY CLAIMS BY THIRD PARTIES
(INCLUDING BUT NOT LIMITED TO ANY DEFENSE THEREOF), OR OTHER SIMILAR COSTS.
*/
#endif /* EEPROM_I2C_H */
EXAMPLE C-6: SAMPLE MAIN FILE CALLING I^2C FUNCTIONS

/**
 * Generated Main Source File
 * Company:
 *     Microchip Technology Inc.
 * File Name:
 *     main.c
 * Summary:
 *     This is the main file generated using MPLAB® Code Configurator
 * Description:
 *     This header file provides implementations for driver APIs for all modules selected in the
 *     GUI.
 * Generation Information :
 *     Product Revision : MPLAB® Code Configurator - v2.25.2
 *     Device            : PIC16F1719
 *     Driver Version    : 2.00
 *     The generated drivers are tested against the following:
 *     Compiler          : XC8 v1.34
 *     MPLAB             : MPLAB X v2.35 or v3.00
 */

/*
Copyright (c) 2013 - 2015 released Microchip Technology Inc. All rights reserved.

Microchip licenses to you the right to use, modify, copy and distribute Software only when embedded on a Microchip microcontroller or digital signal controller that is integrated into your product or third party product (pursuant to the sublicense terms in the accompanying license agreement).

You should refer to the license agreement accompanying this Software for additional information regarding your rights and obligations.

SOFTWARE AND DOCUMENTATION ARE PROVIDED "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING WITHOUT LIMITATION, ANY WARRANTY OF MERCHANTABILITY, TITLE, NON-INFRINGEMENT AND FITNESS FOR A PARTICULAR PURPOSE.
IN NO EVENT SHALL MICROCHIP OR ITS LICENSORS BE LIABLE OR OBLIGATED UNDER CONTRACT, NEGLIGENCE, STRICT LIABILITY, CONTRIBUTION, BREACH OF WARRANTY, OR OTHER LEGAL EQUITABLE THEORY ANY DIRECT OR INDIRECT DAMAGES OR EXPENSES INCLUDING BUT NOT LIMITED TO ANY INCIDENTAL, SPECIAL, INDIRECT, PUNITIVE OR CONSEQUENTIAL DAMAGES, LOST PROFITS OR LOST DATA, COST OF PROCUREMENT OF SUBSTITUTE GOODS, TECHNOLOGY, SERVICES, OR ANY CLAIMS BY THIRD PARTIES (INCLUDING BUT NOT LIMITED TO ANY DEFENSE THEREOF), OR OTHER SIMILAR COSTS.
EXAMPLE C-6: SAMPLE MAIN FILE CALLING I2C FUNCTIONS (CONTINUED)

/*
#include "mcc_generated_files/mcc.h"
#include "eeprom_i2c.h"

/*
Main application
*/

void main(void) {
    // initialize the device
    SYSTEM_Initialize();

    // When using interrupts, you need to set the Global and Peripheral Interrupt Enable bits
    // Use the following macros to:
    // Enable the Global Interrupts
    INTERRUPT_GlobalInterruptEnable();

    // Enable the Peripheral Interrupts
    INTERRUPT_PeripheralInterruptEnable();

    // Disable the Global Interrupts
    INTERRUPT_GlobalInterruptDisable();

    // Disable the Peripheral Interrupts
    INTERRUPT_PeripheralInterruptDisable();

    uint8_t sourceData[] = {0x1A, 0x2A, 0x4A, 0x8A, 0x1A, 0x2A, 0x4A, 0x8A, 0x1A, 0x2A, 0x4A, 0x8A, 0x1A, 0x2A, 0x4A, 0x8A, 0x1A, 0x2A, 0x4A, 0x8A};
    uint8_t addressBuffer[] = {0xAB, 0x10} ; //Put your address here
    uint8_t readBuffer[16];
    uint8_t readByte;

    int r = 0;

    //Writes a byte of data to address specified
    r = I2C_ByteWrite(&addressBuffer,0x5B,sizeof(addressBuffer));

    //Reads a byte of data stored at the address specified
    I2C_ByteRead(&addressBuffer,&readByte,sizeof(addressBuffer));

    //Write a specified number of data bytes beginning at the specified address
    r = I2C_BufferWrite(&addressBuffer,&sourceData,sizeof(addressBuffer),4);

    //Reads a specified number of data bytes beginning at the specified address
    I2C_BufferRead(&addressBuffer,&readBuffer,sizeof(addressBuffer),4);

    //stop here
    while (1) {
        // Add your application code
    }
}

/**
End of File
*/
Note the following details of the code protection feature on Microchip devices:

- Microchip products meet the specification contained in their particular Microchip Data Sheet.
- Microchip believes that its family of products is one of the most secure families of its kind on the market today, when used in the intended manner and under normal conditions.
- There are dishonest and possibly illegal methods used to breach the code protection feature. All of these methods, to our knowledge, require using the Microchip products in a manner outside the operating specifications contained in Microchip’s Data Sheets. Most likely, the person doing so is engaged in theft of intellectual property.
- Microchip is willing to work with the customer who is concerned about the integrity of their code.
- Neither Microchip nor any other semiconductor manufacturer can guarantee the security of their code. Code protection does not mean that we are guaranteeing the product as “unbreakable.”

Code protection is constantly evolving. We at Microchip are committed to continuously improving the code protection features of our products. Attempts to break Microchip’s code protection feature may be a violation of the Digital Millennium Copyright Act. If such acts allow unauthorized access to your software or other copyrighted work, you may have a right to sue for relief under that Act.

Information contained in this publication regarding device applications and the like is provided only for your convenience and may be superseded by updates. It is your responsibility to ensure that your application meets with your specifications. MICROCHIP MAKES NO REPRESENTATIONS OR WARRANTIES OF ANY KIND WHETHER EXPRESS OR IMPLIED, WRITTEN OR ORAL, STATUTORY OR OTHERWISE, RELATED TO THE INFORMATION, INCLUDING BUT NOT LIMITED TO ITS CONDITION, QUALITY, PERFORMANCE, MERCHANTABILITY OR FITNESS FOR PURPOSE. Microchip disclaims all liability arising from this information and its use. Use of Microchip devices in life support and/or safety applications is entirely at the buyer’s risk, and the buyer agrees to defend, indemnify and hold harmless Microchip from any and all damages, claims, suits, or expenses resulting from such use. No licenses are conveyed, implicitly or otherwise, under any Microchip intellectual property rights unless otherwise stated.

Trademarks

The Microchip name and logo, the Microchip logo, dsPIC, FlashFlex, flexPWR, JukeBlox, KEELoo, KEELoo logo, Kleer, LANCheck, MediaLB, MOST, MOST logo, MPLAB, OptoLyzer, PIC, PICSTART, PIC® logo, RightTouch, SpyNIC, SST, SST Logo, SuperFlash and UNI/O are registered trademarks of Microchip Technology Incorporated in the U.S.A. and other countries.

The Embedded Control Solutions Company and mTouch are registered trademarks of Microchip Technology Incorporated in the U.S.A.


SQTP is a service mark of Microchip Technology Incorporated in the U.S.A.

Silicon Storage Technology is a registered trademark of Microchip Technology Inc. in other countries.

GestIC is a registered trademark of Microchip Technology Germany II GmbH & Co. KG, a subsidiary of Microchip Technology Inc., in other countries.

All other trademarks mentioned herein are property of their respective companies.

© 2016, Microchip Technology Incorporated, Printed in the U.S.A., All Rights Reserved.


QUALITY MANAGEMENT SYSTEM CERTIFIED BY DNV

ISO/TS 16949

Microchip received ISO/TS-16949:2009 certification for its worldwide headquarters, design and wafer fabrication facilities in Chandler and Tempe, Arizona; Gresham, Oregon and design centers in California and India. The Company’s quality system processes and procedures are for its PIC® MCUs and dsPIC® DSCs, dsPIC® code hopping devices, Serial EEPROMs, microperipherals, nonvolatile memory and analog products. In addition, Microchip’s quality system for the design and manufacture of development systems is ISO 9001:2000 certified.
# Worldwide Sales and Service

## AMERICAS
**Corporate Office**
2355 West Chandler Blvd.
Chandler, AZ 85224-6199
Tel: 480-792-7200
Fax: 480-792-7277
Technical Support: [http://www.microchip.com/support](http://www.microchip.com/support)

**Atlanta**
Duluth, GA
Tel: 678-957-9614
Fax: 678-957-1455

**Austin, TX**
Tel: 512-257-3370

**Boston**
Westborough, MA
Tel: 774-760-0087
Fax: 774-760-0088

**Chicago**
Itasca, IL
Tel: 630-285-0071
Fax: 630-285-0075

**Cleveland**
Independence, OH
Tel: 216-447-0464
Fax: 216-447-0643

**Dallas**
Addison, TX
Tel: 972-818-7423
Fax: 972-818-2924

**Detroit**
Novi, MI
Tel: 248-848-4000

**Houston, TX**
Tel: 281-894-5983

**Indianapolis**
Noblesville, IN
Tel: 317-773-8323
Fax: 317-773-5453

**Los Angeles**
Mission Viejo, CA
Tel: 949-462-9523
Fax: 949-462-9608

**New York, NY**
Tel: 631-435-6000

**San Jose, CA**
Tel: 408-735-9110

**Canada - Toronto**
Tel: 905-673-0699
Fax: 905-673-6509

## ASIA/PACIFIC
**Asia Pacific Office**
Suites 3707-14, 37th Floor
Tower 6, The Gateway
Harbour City, Kowloon

**Hong Kong**
Tel: 852-2943-5100
Fax: 852-2401-3431

**Australia - Sydney**
Tel: 61-2-9868-6733
Fax: 61-2-9868-6755

**China - Beijing**
Tel: 86-10-8569-7000
Fax: 86-10-8528-2104

**China - Chengdu**
Tel: 86-28-8665-5511
Fax: 86-28-8665-7889

**China - Chongqing**
Tel: 86-23-8960-9588
Fax: 86-23-8960-9500

**China - Dongguan**
Tel: 86-769-8702-9880
Fax: 86-769-8702-9880

**China - Hangzhou**
Tel: 86-571-8792-8115
Fax: 86-571-8792-8116

**China - Hong Kong SAR**
Tel: 852-2943-5100
Fax: 852-2401-3431

**China - Nanjing**
Tel: 86-25-8473-2460
Fax: 86-25-8473-2470

**China - Qingdao**
Tel: 86-632-8502-7355
Fax: 86-632-8502-7205

**China - Shanghai**
Tel: 86-21-5407-5533
Fax: 86-21-5407-5066

**China - Shenyang**
Tel: 86-24-2334-2829
Fax: 86-24-2334-2393

**China - Shenzhen**
Tel: 86-755-8864-2200
Fax: 86-755-8203-1760

**China - Wuhan**
Tel: 86-27-5980-5300
Fax: 86-27-5980-5118

**China - Xian**
Tel: 86-29-8833-7252
Fax: 86-29-8833-7256

**India - Bangalore**
Tel: 91-80-3090-4444
Fax: 91-80-3090-4444

**India - New Delhi**
Tel: 91-11-4160-8631
Fax: 91-11-4160-8632

**India - Pune**
Tel: 91-20-3019-1500

**Japan - Osaka**
Tel: 81-6-6152-7160
Fax: 81-6-6152-9310

**Japan - Tokyo**
Tel: 81-3-6880-3770
Fax: 81-3-6880-3771

**Korea - Daegu**
Tel: 82-53-744-4301
Fax: 82-53-744-4302

**Korea - Seoul**
Tel: 82-2-554-7200
Fax: 82-2-558-5932 or 82-2-558-5934

**Malaysia - Kuala Lumpur**
Tel: 60-3-6201-9857
Fax: 60-3-6201-9859

**Malaysia - Penang**
Tel: 60-4-227-8870
Fax: 60-4-227-4068

**Philippines - Manila**
Tel: 63-2-634-9065
Fax: 63-2-634-9069

**Singapore**
Tel: 65-6334-8870
Fax: 65-6334-8850

**Taiwan - Hsin Chu**
Tel: 886-3-577-366
Fax: 886-3-577-995

**Taiwan - Kaohsiung**
Tel: 886-7-213-7828

**Taiwan - Taipei**
Tel: 886-2-2308-6600
Fax: 886-2-2308-0102

**Thailand - Bangkok**
Tel: 66-2-694-1351
Fax: 66-2-694-1350

## ASIA/PACIFIC
**China - Xiamen**
Tel: 86-592-2388138
Fax: 86-592-2388130

**China - Zhuhai**
Tel: 86-756-3210040
Fax: 86-756-3210049

## EUROPE
**Austria - Wels**
Tel: 43-7242-2244-39
Fax: 43-7242-2244-39

**Denmark - Copenhagen**
Tel: 45-4450-2828
Fax: 45-4485-2829

**France - Paris**
Tel: 33-1-69-53-63-20
Fax: 33-1-69-30-90-79

**Germany - Dusseldorf**
Tel: 49-2129-3766400

**Germany - Karlsruhe**
Tel: 49-721-625370

**Germany - Munich**
Tel: 49-89-627-144-0
Fax: 49-89-627-144-44

**Italy - Milan**
Tel: 39-0331-742611
Fax: 39-0331-466781

**Italy - Venice**
Tel: 39-049-7625286

**Netherlands - Drunen**
Tel: 31-416-690399
Fax: 31-416-690340

**Poland - Warsaw**
Tel: 48-22-3325737

**Spain - Madrid**
Tel: 34-91-708-08-90
Fax: 34-91-708-08-91

**Sweden - Stockholm**
Tel: 46-8-5090-4654

**UK - Wokingham**
Tel: 44-118-921-5800
Fax: 44-118-921-5820

---

07/14/15