MICROCHIP

TB3281

Getting Started with I²C Using MSSP on PIC18

Introduction

Author: Filip Manole, Microchip Technology Inc.

The approach in implementing the I²C communication protocol is different among the PIC18F device family of microcontrollers. While the PIC18-K40 and PIC18-Q10 product families have a Master Synchronous Serial Port (MSSP) peripheral, the PIC18-K42, PIC18-K83, PIC18-Q41, PIC18-Q43 and PIC18-Q84 product families have a dedicated I²C peripheral.

The MSSP and I²C peripherals are serial interfaces useful for communicating with external hardware, such as sensors or microcontroller devices, but there are also differences between them. The MSSP peripheral can operate in one of two modes: Serial Peripheral Interface (SPI) and Inter-Integrated Circuit (I²C), having the advantage of implementing both communication protocols with the same hardware. For a detailed comparison between the MSSP and dedicated I²C peripherals, refer to: Master Synchronous Serial Port (MSSP) to the Stand-Alone I²C Module Migration.

This technical brief provides information about the MSSP peripheral of the PIC18-K40 and PIC18-Q10 product families and intends to familiarize the user with the PIC® microcontrollers. The document covers the following use cases:

- **Master Write Data:**
  This example shows how the microcontroller configured in I²C Master mode writes data to an MCP23008 8-bit I²C I/O expander (slave device), addressed in 7-bit mode.
- **Master Read/Write Data Using Interrupts:**
  This example shows how the microcontroller configured in I²C Master mode writes to and reads data from an MCP23008 8-bit I²C I/O expander (slave device), addressed in 7-bit mode, using interrupts.

For each use case, there are three different implementations, which have the same functionalities: one code generated with MPLAB® Code Configurator (MCC), one code generated using Foundation Services Library, and one bare metal code. The MCC generated code offers hardware abstraction layers that ease the use of the code across different devices from the same family. The Foundation Services generated code offers a driver-independent Application Programming Interface (API), and facilitates the portability of code across different platforms. The bare metal code is easier to follow, allowing a fast ramp-up on the use case associated code.

Note: The examples in this technical brief have been developed using PIC18F47Q10 Curiosity Nano development board. The PIC18F47Q10 pin package present on the board is QFN.

Table of Contents

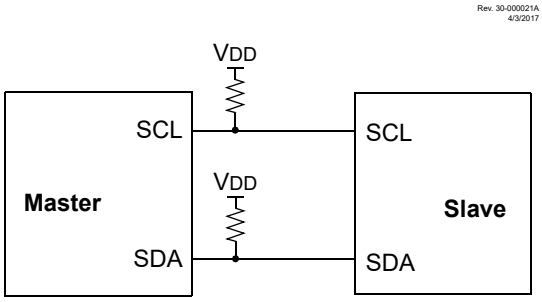
1. Peripheral Overview

The I²C bus is a multi-master serial data communication bus. Microcontrollers communicate in a master/slave environment where the master devices initiate the communication and the devices are selected through addressing.

I²C operates with one or more master devices and one or more slave devices. A given device can operate in four modes:

- Master Transmit mode – master is transmitting data to a slave
- Master Receive mode – master is receiving data from a slave
- Slave Transmit mode – slave is transmitting data to a master
- Slave Receive mode – slave is receiving data from a master

Figure 1-1. I²C Master/Slave Connection

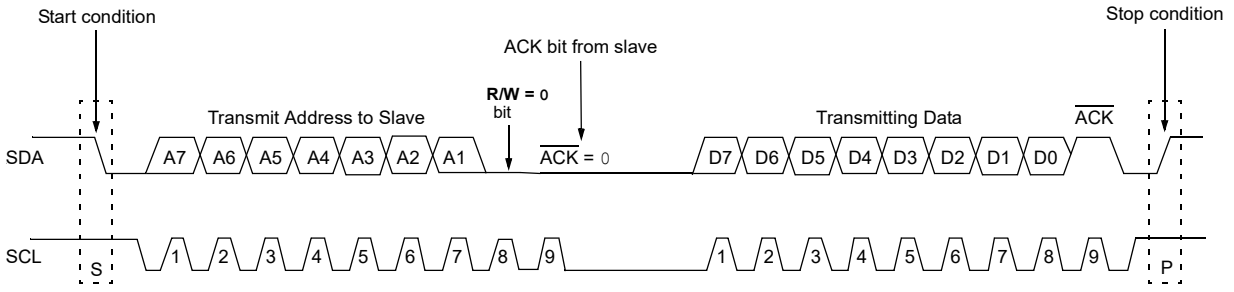


To begin communication, the master device sends out a Start bit followed by the address byte of the slave it intends to communicate with. This is followed by a bit which determines if the master intends to write to or read from the slave.

If there is a slave on the bus with the indicated address, it will respond with an Acknowledge bit. After the master receives the Acknowledge bit, it can continue in either Writing or Reading mode.

- If the master intends to write to the slave, then it will send a byte and wait for an Acknowledge bit for each sent byte.
- If the master intends to read from the slave, then it will receive a byte and respond with an Acknowledge bit for each received byte.

Figure 1-2. I²C Transmission



I²C protocol:

- The Start bit is indicated by a high-to-low transition of the SDA line while the SCL line is held high
- The Acknowledge bit is an active-low signal, which holds the SDA line low to indicate to the transmitter that the slave device has received the transmitted data and is ready to receive more
- A transition of a data bit is always performed while the SCL line is held low. Transitions that occur while the SCL line is held high are used to indicate Start and Stop bits.

The MSSP registers used to configure the device in I^2C Master mode:

- MSSP Control register 1 (SSPxCON1) used to enable the MSSP peripheral and set the device in I^2C Master mode
- MSSP Control register 2 (SSPxCON2) used to send the Start and Stop conditions, set the Receive mode and handle the Acknowledge bits
- MSSP Data Buffer (SSPxBUF) register used to send the bytes to and receive the bytes from the slave
- In addition, this is the address the I^2C slave responds to

```
I2C Clock = F_OSC / (4 * (SSP1ADD + 1)
```

2. Master Write Data

In this use case, the microcontroller is configured in I²C Master mode using the MSSP1 instance of the MSSP peripheral, and communicates with the slave MCP23008, an 8-bit I/O expander that can be controlled through the I²C interface.

The extended pins are set as digital output with an I²C write operation in the slave's I/O Direction (IODIR) register.

After the pins are set, the program will repeatedly:

- set pins to digital low, with an I²C write operation in the GPIO register;
- set pins to digital high, with an I²C write operation in the GPIO register.

To transmit data as master, the following sequence must be implemented:

1. Generate the Start condition by setting the SEN bit in the SSPxCON2 register.
2. The SSPxIF flag in the PIR3 register is set by hardware on completion of the Start condition, and must be cleared by software.
3. Load the slave address in the SSPxBUF register.
4. The SSPxIF flag in the PIR3 register is set by hardware and must be cleared by software.
5. Check the ACKSTAT bit in the SSPxCON2 register.
6. Load the register address in the SSPxBUF register.
7. The SSPxIF flag in the PIR3 register is set by hardware and must be cleared by software.
8. Check the ACKSTAT bit in the SSPxCON2 register.
9. Load the data in the SSPxBUF register.
10. The SSPxIF flag in the PIR3 register is set by hardware and must be cleared by software.
11. Check the ACKSTAT bit in the SSPxCON2 register.
12. To end the transmission, generate the Stop condition by setting the PEN bit in the SSPxCON2 register.
13. The SSPxIF flag in the PIR3 register is set by hardware and must be cleared by software.

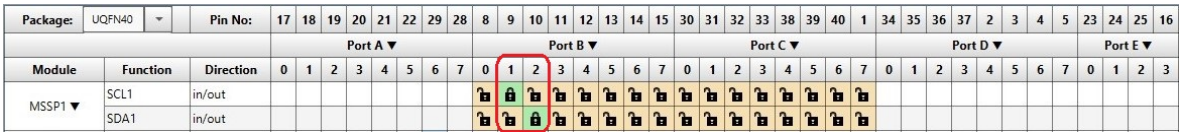
Note: For a reliable I²C operation, external pull-up resistors must be added. Refer to TB3191 - I2C Master Mode for more details.

2.1 MCC Generated

To generate this project using MPLAB® Code Configurator (MCC), follow the next steps:

1. Create a new MPLAB X IDE project for PIC18F47Q10.
2. Open MCC from the toolbar (more information on how to install the MCC plug-in can be found here).
3. Go to *Project Resources → System → System Module* and do the following configuration:
   - Oscillator Select: HFINTOSC
   - HF Internal Clock: 64 MHz
   - Clock Divider: 1
   - In the Watchdog Timer Enable field in the **WWDT** tab, **WDT Disabled** has to be selected.
   - In the **Programming** tab, **Low-Voltage Programming Enable** has to be checked.
4. From the Device Resources window, add MSSP1, then do the following configuration:
   - Serial Protocol: I²C
   - Mode: Master
   - I²C Clock Frequency: 100000
5. Open the *Pin Manager → Grid View* window, select **UQFN40** in the MCU package field, and do the following pin configurations:

Figure 2-1. Pin Mapping



6. Go to *Project Resources → Pin Module* and set both pins, RB1 and RB2, to use the internal pull-up by checking the box in the WPU column. Ensure that for MSSP1, SCL is assigned to pin RB1 and SDA is assigned to RB2 in the pin manager grid view.
7. In the Project Resources window, click the **Generate** button so that MCC will generate all the specified drivers and configurations.
8. Edit the `main.c` file, as following:

```
#include "mcc_generated_files/mcc.h"
#include "mcc_generated_files/examples/i2c1_master_example.h"

#define I2C_SLAVE_ADDR              0x20
#define MCP23008_REG_ADDR_IODIR     0x00
#define MCP23008_REG_ADDR_GPIO      0x09
#define PINS_DIGITAL_OUTPUT         0x00
#define PINS_DIGITAL_LOW            0x00
#define PINS_DIGITAL_HIGH           0xFF

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

    /* Set the extended pins as digital output */
    I2C1_Write1ByteRegister(I2C_SLAVE_ADDR, MCP23008_REG_ADDR_IODIR, PINS_DIGITAL_OUTPUT);

    while (1)
    {
        /* Set the extended pins to digital low */
        I2C1_Write1ByteRegister(I2C_SLAVE_ADDR, MCP23008_REG_ADDR_GPIO, PINS_DIGITAL_LOW);
        __delay_ms(500);
        /* Set the extended pins to digital high */
        I2C1_Write1ByteRegister(I2C_SLAVE_ADDR, MCP23008_REG_ADDR_GPIO, PINS_DIGITAL_HIGH);
        __delay_ms(500);
    }
}
```

View the PIC18F47Q10 Code Example on GitHub
Click to browse repositories

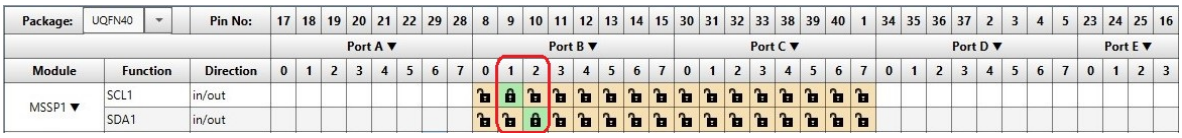
2.2 Foundation Services

To generate this project using Foundation Services (FS), follow the next steps:

1. Create a new MPLAB X IDE project for PIC18F47Q10.
2. Open MCC from the toolbar (more information on how to install the MCC plug-in can be found here).
3. Go to *Project Resources → System → System Module* and do the following configuration:
   - Oscillator Select: HFINTOSC
   - HF Internal Clock: 64 MHz
   - Clock Divider: 1
   - In the Watchdog Timer Enable field in the **WWDT** tab, **WDT Disabled** has to be selected.
   - In the **Programming** tab, **Low-Voltage Programming Enable** has to be checked.
4. From the Device Resources window, add I2CSIMPLE, then do the following configuration:
   - Select **I²C Master**: MSSP1

5. Open the *Pin Manager → Grid View* window, select **UQFN40** in the MCU package field, and do the following pin configurations:

Figure 2-2. Pin Mapping



6. Go to *Project Resources → Pin Module* and set both pins, RB1 and RB2, to use the internal pull-up by checking the box in the WPU column.
7. In the Project Resources window, click the **Generate** button so that MCC will generate all the specified drivers and configurations.
8. Edit the `main.c` file, as following:

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

#define I2C_SLAVE_ADDR              0x20
#define MCP23008_REG_ADDR_IODIR     0x00
#define MCP23008_REG_ADDR_GPIO      0x09
#define PINS_DIGITAL_OUTPUT         0x00
#define PINS_DIGITAL_LOW            0x00
#define PINS_DIGITAL_HIGH           0xFF

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

    /* Set the extended pins as digital output */
    i2c_write1ByteRegister(I2C_SLAVE_ADDR, MCP23008_REG_ADDR_IODIR, PINS_DIGITAL_OUTPUT);

    while (1)
    {
        /* Set the extended pins to digital low */
        i2c_write1ByteRegister(I2C_SLAVE_ADDR, MCP23008_REG_ADDR_GPIO, PINS_DIGITAL_LOW);
        __delay_ms(500);
        /* Set the extended pins to digital high */
        i2c_write1ByteRegister(I2C_SLAVE_ADDR, MCP23008_REG_ADDR_GPIO, PINS_DIGITAL_HIGH);
        __delay_ms(500);
    }
}
```

View the PIC18F47Q10 Code Example on GitHub
Click to browse repositories

2.3 Bare Metal Code

The necessary code and functions to implement the presented example are analyzed in this section.

The first step will be to configure the microcontroller to disable the Watchdog Timer and to enable the Low-Voltage Programming (LVP).

```
#pragma config WDTE = OFF      /* WDT operating mode → WDT Disabled */
#pragma config LVP = ON        /* Low-voltage programming enabled, RE3 pin is MCLR */
```

Define the `_XTAL_FREQ` to the clock frequency and include the used libraries.

```
#define _XTAL_FREQ                  64000000UL
#include <pic18.h>
#include <xc.h>
#include <stdint.h>
```

The `CLK_Initialize` function selects the oscillator and the clock divider, and sets the nominal frequency:

```
static void CLK_Initialize(void)
{
    /* Set Oscillator Source: HFINTOSC and Set Clock Divider: 1 */
    OSCCON1bits.NOSC = 0x6;

    /* Set Nominal Freq: 64 MHz */
    OSCFRQbits.FRQ3 = 1;
}
```

The `PPS_Initialize` function routes the SCL to pin RB1 and SDA to pin RB2:

```
static void PPS_Initialize(void)
{
    /* PPS setting for using RB1 as SCL */
    SSP1CLKPPS = 0x09;
    RB1PPS = 0x0F;

    /* PPS setting for using RB2 as SDA */
    SSP1DATPPS = 0x0A;
    RB2PPS = 0x10;
}
```

The `PORT_Initialize` function sets pins, RB1 and RB2, as digital pins with internal pull-up resistors:

```
static void PORT_Initialize(void)
{
    /* Set pins RB1 and RB2 as Digital */
    ANSELBbits.ANSELB1 = 0;
    ANSELBbits.ANSELB2 = 0;

    /* Set pull-up resistors for RB1 and RB2 */
    WPUBbits.WPUB1 = 1;
    WPUBbits.WPUB2 = 1;
}
```

The `I2C1_Initialize` function selects the I^2C Master mode and sets the I^2C clock frequency to 100 kHz:

```
static void I2C1_Initialize(void)
{
    /* I2C Master Mode: Clock = F_OSC / (4 * (SSP1ADD + 1)) */
    SSP1CON1bits.SSPM3 = 1;

    /* Set the baud rate divider to obtain the I2C clock at 100000 Hz*/
    SSP1ADD  = 0x9F;
}
```

The `I2C1_interruptFlagPolling` function waits for the SSP1IF flag to be triggered by the hardware and clears it:

```
static void I2C1_interruptFlagPolling(void)
{
    /* Polling Interrupt Flag */

    while (!PIR3bits.SSP1IF)
    {
        ;
    }

    /* Clear the Interrupt Flag */
    PIR3bits.SSP1IF = 0;
}
```

The `I2C1_open` function prepares an I²C operation: Resets the SSP1IF flag and enables the SSP1 module:

```
static void I2C1_open(void)
{
    /* Clear IRQ */
    PIR3bits.SSP1IF = 0;

    /* I2C Master Open */
    SSP1CON1bits.SSPEN = 1;
}
```

The `I2C1_close` function disables the SSP1 module:

```
static void I2C1_close(void)
{
    /* Disable I2C1 */
    SSP1CON1bits.SSPEN = 0;
}
```

The `I2C1_start` function sends the Start bit by setting the SEN bit and waits for the SSP1IF flag to be triggered:

```
static void I2C1_startCondition(void)
{
    /* Start Condition*/
    SSP1CON2bits.SEN = 1;
    I2C1_interruptFlagPolling();
}
```

The `I2C1_stop` function sends the Stop bit and waits for the SSP1IF flag to be triggered:

```
static void I2C1_stopCondition(void)
{
    /* Stop Condition */
    SSP1CON2bits.PEN = 1;
    I2C1_interruptFlagPolling();
}
```

The `I2C1_sendData` function loads in SSP1BUF the argument value and waits for the SSP1IF flag to be triggered:

```
static void I2C1_sendData(uint8_t byte)
{
    SSP1BUF  = byte;
    I2C1_interruptFlagPolling();
}
```

The `I2C1_getAckstatBit` function returns the ACKSTAT bit from the SSP1CON2 register:

```
static uint8_t I2C1_getAckstatBit(void)
{
    /* Return ACKSTAT bit */
    return SSP1CON2bits.ACKSTAT;
}
```

The `I2C1_write1ByteRegister` function executes all the steps to write one byte to the slave:

```
static void I2C1_write1ByteRegister(uint8_t address, uint8_t reg, uint8_t data)
{
    /* Shift the 7-bit address and add a 0 bit to indicate a write operation */
    uint8_t writeAddress = (address << 1) & ~I2C_RW_BIT;

    I2C1_open();
    I2C1_start();

    I2C1_sendData(writeAddress);
    if (I2C1_getAckstatBit())
    {
        return ;
    }
```

```
    I2C1_sendData(reg);
    if (I2C1_getAckstatBit())
    {
        return ;
    }

    I2C1_sendData(data);
    if (I2C1_getAckstatBit())
    {
        return ;
    }

    I2C1_stop();
    I2C1_close();
}
```

The `main` function has multiple responsibilities:

- Initializes the clock frequency, peripheral pin select, ports and I^2C peripheral.
- Sets the slave's I/O Direction (IODIR) register to '0', the value for digital output pins.
- Continuously sets the slave's General Purpose I/O PORT register to digital low and digital high using I^2C write operations.

```
#define I2C_SLAVE_ADDR                  0x20
#define MCP23008_REG_ADDR_IODIR         0x00
#define MCP23008_REG_ADDR_GPIO          0x09
#define PINS_DIGITAL_OUTPUT             0x00
#define PINS_DIGITAL_LOW                0x00
#define PINS_DIGITAL_HIGH               0xFF

void main(void)
{
    CLK_Initialize();
    PPS_Initialize();
    PORT_Initialize();
    I2C1_Initialize();

    /* Set the extended pins as digital output */
    I2C1_write1ByteRegister(I2C_SLAVE_ADDR, MCP23008_REG_ADDR_IODIR, PINS_DIGITAL_OUTPUT);

    while (1)
    {
        /* Set the extended pins to digital low */
        I2C1_write1ByteRegister(I2C_SLAVE_ADDR, MCP23008_REG_ADDR_GPIO, PINS_DIGITAL_LOW);
        __delay_ms(500);
        /* Set the extended pins to digital high */
        I2C1_write1ByteRegister(I2C_SLAVE_ADDR, MCP23008_REG_ADDR_GPIO, PINS_DIGITAL_HIGH);
        __delay_ms(500);
    }
}
```

View the PIC18F47Q10 Code Example on GitHub
Click to browse repositories

3. Master Read/Write Data Using Interrupts

In this use case, the microcontroller is configured in I²C Master mode using the MSSP1 instance of the MSSP peripheral, and communicates with the slave MCP23008, an 8-bit I/O expander that can be controlled through the I²C interface.

The extended pins are set as digital outputs with an I²C write operation in the slave's I/O Direction (IODIR) register.

After the pins are set, the program will repeatedly:

- set the pins to the value of the `data` variable, with an I²C write operation in the GPIO register;
- read from the GPIO register, with an I²C read operation and save the value into the `data` variable;
- invert the bits in the `data` variable to write another value into the GPIO register in the next loop.

To read data from the MCP23008 device, the following sequence must be implemented:

1. Generate the Start condition by setting the SEN bit in the SSPxCON2 register.
2. The SSPxIF flag in the PIR3 register is set by hardware on completion of the Start condition and must be cleared by software.
3. Load the slave address in the SSPxBUF register.
4. The SSPxIF flag in the PIR3 register is set by hardware and must be cleared by software.
5. Check the ACKSTAT bit in the SSPxCON2 register.
6. Load the register address in the SSPxBUF register.
7. The SSPxIF flag in the PIR3 register is set by hardware and must be cleared by software.
8. Check the ACKSTAT bit in the SSPxCON2 register.
9. Generate the Start condition by setting the SEN bit.
10. The SSPxIF flag in the PIR3 register is set by hardware on completion of the Start condition and must be cleared by software.
11. Load the slave address in the SSPxBUF register.
12. The SSPxIF flag in the PIR3 register is set by hardware and must be cleared by software.
13. Check the ACKSTAT bit in the SSPxCON2 register.
14. Set the RCEN bit to enable the Receive mode.
15. The SSPxIF flag in the PIR3 register is set by hardware and must be cleared by software.
16. Read data from the SSPxBUF register.
17. Send a Not Acknowledge bit to stop receiving bytes.
18. To end the transmission, generate the Stop condition by setting the PEN bit in the SSPxCON2 register.
19. The SSPxIF flag in the PIR3 register is set by hardware and must be cleared by software.

Note: For a reliable I²C operation, external pull-up resistors must be added. Refer to TB3191 - I2C Master Mode for more details.

3.1 MCC Generated

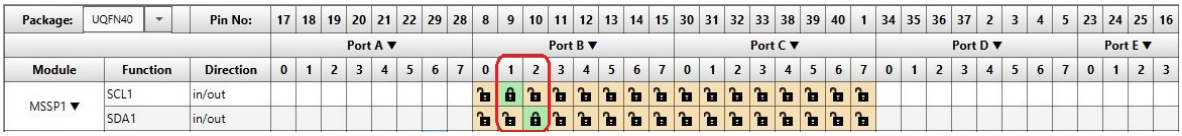
To generate this project using MPLAB Code Configurator (MCC), follow the next steps:

1. Create a new MPLAB X IDE project for PIC18F47Q10.
2. Open MCC from the toolbar (more information on how to install the MCC plug-in can be found here).
3. Go to *Project Resources → System → System Module* and do the following configuration:
   - Oscillator Select: HFINTOSC
   - HF Internal Clock: 64 MHz
   - Clock Divider: 1
   - In the Watchdog Timer Enable field in the **WWDT** tab, **WDT Disabled** has to be selected
   - In the **Programming** tab, **Low-Voltage Programming Enable** has to be checked
4. From the Device Resources window, add MSSP1, then do the following configuration:
   - Interrupt Driven: Checked

- Serial Protocol: I²C
- Mode: Master
- I²C Clock Frequency: 100000

5. Open the *Pin Manager → Grid View* window, select **UQFN40** in the MCU package field, and do the following pin configurations:

Figure 3-1. Pin Mapping



6. Go to *Project Resources → Pin Module* and set both pins, RB1 and RB2, to use the internal pull-up by checking the box in the WPU column.
7. In the Project Resources window, click the **Generate** button so that MCC will generate all the specified drivers and configurations.
8. Edit the `main.c` file, as following:

```
#include "mcc_generated_files/mcc.h"
#include "mcc_generated_files/examples/i2c1_master_example.h"


#define I2C_SLAVE_ADDR              0x20
#define MCP23008_REG_ADDR_IODIR     0x00
#define MCP23008_REG_ADDR_GPIO      0x09
#define PINS_DIGITAL_OUTPUT         0x00
#define MCP23008_DATA               0x0F

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

    // Enable the Global Interrupts
    INTERRUPT_GlobalInterruptEnable();

    // Enable the Peripheral Interrupts
    INTERRUPT_PeripheralInterruptEnable();

    /* Set data to use in the I2C operations */
    uint8_t data = MCP23008_DATA;
    /* Set the extended pins as digital output */
    I2C1_Write1ByteRegister(I2C_SLAVE_ADDR, MCP23008_REG_ADDR_IODIR, PINS_DIGITAL_OUTPUT);

    while (1)
    {
        /* Write data to the GPIO port */
        I2C1_Write1ByteRegister(I2C_SLAVE_ADDR, MCP23008_REG_ADDR_GPIO, data);
        /* Read data from the GPIO port */
        data = I2C1_Read1ByteRegister(I2C_SLAVE_ADDR, MCP23008_REG_ADDR_GPIO);
        /* Overwrite data with the inverted data read */
        data = ~data;

        __delay_ms(500);
    }
}
```

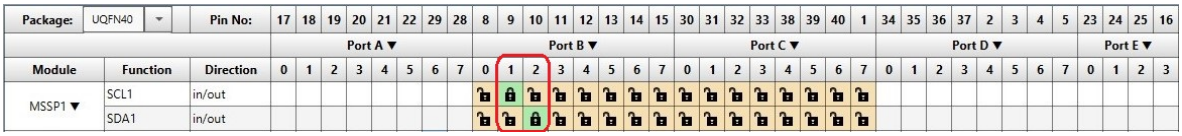
View the PIC18F47Q10 Code Example on GitHub
Click to browse repositories

3.2 Foundation Services

To generate this project using Foundation Services (FS), follow the next steps:

1. Create a new MPLAB X IDE project for PIC18F47Q10.
2. Open MCC from the toolbar (more information on how to install the MCC plug-in can be found here).
3. Go to *Project Resources → System → System Module* and do the following configuration:
   – Oscillator Select: HFINTOSC
   – HF Internal Clock: 64 MHz
   – Clock Divider: 1
   – In the Watchdog Timer Enable field in the **WWDT** tab, **WDT Disabled** has to be selected
   – In the **Programming** tab, **Low-Voltage Programming Enable** has to be checked
4. From the Device Resources window, add I2CSIMPLE, then do the following configuration:
   – Select **I²C Master**: MSSP1
5. Go to *Device Resources → Peripherals → MSSP1*, then check the **Interrupt Driven** box.
6. Open the *Pin Manager → Grid View* window, select **UQFN40** in the MCU package field and do the following pin configurations:

Figure 3-2. Pin Mapping



7. Go to *Project Resources → Pin Module*, and set both pins, RB1 and RB2, to use the internal pull-up by checking the box in the WPU column.
8. In the Project Resources window, click the **Generate** button so that MCC will generate all the specified drivers and configurations.
9. Edit the `main.c` file, as following:

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

#define I2C_SLAVE_ADDR              0x20
#define MCP23008_REG_ADDR_IODIR     0x00
#define MCP23008_REG_ADDR_GPIO      0x09
#define PINS_DIGITAL_OUTPUT         0x00
#define MCP23008_DATA               0x0F

/*
                         Main application
 */
void main(void)
{
    // Initialize the device
    SYSTEM_Initialize();

    // Enable the Global Interrupts
    INTERRUPT_GlobalInterruptEnable();

    // Enable the Peripheral Interrupts
    INTERRUPT_PeripheralInterruptEnable();

    /* Set data to use in the I2C operations */
    uint8_t data = MCP23008_DATA;
    /* Set the extended pins as digital output */
    i2c_write1ByteRegister(I2C_SLAVE_ADDR, MCP23008_REG_ADDR_IODIR, PINS_DIGITAL_OUTPUT);

    while (1)
    {
        /* Write data to the GPIO port */
        i2c_write1ByteRegister(I2C_SLAVE_ADDR, MCP23008_REG_ADDR_GPIO, data);
        /* Read data from the GPIO port */
        data = i2c_read1ByteRegister(I2C_SLAVE_ADDR, MCP23008_REG_ADDR_GPIO);
        /* Overwrite data with the inverted data read */
        data = ~data;
```

```
        __delay_ms(500);
    }
}
```

View the PIC18F47Q10 Code Example on GitHub
Click to browse repositories

3.3 Bare Metal Code

The necessary code and functions to implement the presented example are analyzed in this section.

The first step will be to configure the microcontroller to disable the Watchdog Timer and to enable the Low-Voltage Programming (LVP).

```
#pragma config WDTE = OFF      /* WDT operating mode → WDT Disabled */
#pragma config LVP = ON        /* Low-voltage programming enabled, RE3 pin is MCLR */
```

The `CLK_Initialize` function selects the HFINTOSC oscillator and the clock divider, and sets the nominal frequency to 64 MHz:

```
static void CLK_Initialize(void)
{
    /* Set Oscillator Source: HFINTOSC and Set Clock Divider: 1 */
    OSCCON1bits.NOSC = 0x6;

    /* Set Nominal Freq: 64 MHz */
    OSCFRQbits.FRQ3 = 1;
}
```

The `PPS_Initialize` function routes the SCL to pin RB1 and SDA to pin RB2:

```
static void PPS_Initialize(void)
{
    /* PPS setting for using RB1 as SCL */
    SSP1CLKPPS = 0x09;
    RB1PPS = 0x0F;

    /* PPS setting for using RB2 as SDA */
    SSP1DATPPS = 0x0A;
    RB2PPS = 0x10;
}
```

The `PORT_Initialize` function sets pins, RB1 and RB2, as digital pins with internal pull-up resistors:

```
static void PORT_Initialize(void)
{
    /* Set pins RB1 and RB2 as Digital */
    ANSELBbits.ANSELB1 = 0;
    ANSELBbits.ANSELB2 = 0;

    /* Set pull-up resistors for RB1 and RB2 */
    WPUBbits.WPUB1 = 1;
    WPUBbits.WPUB2 = 1;
}
```

The `I2C1_Initialize` function selects the I²C Master mode and the baud rate divider:

```
static void I2C1_Initialize(void)
{
    /* I2C Master Mode: Clock = F_OSC / (4 * (SSP1ADD + 1)) */
    SSP1CON1bits.SSPM3 = 1;

    /* Set the baud rate divider to obtain the I2C clock at 100000 Hz*/
    SSP1ADD  = 0x9F;
}
```

The `INTERRUPT_Initialize` function enables the global and peripheral interrupts:

```
static void INTERRUPT_Initialize(void)
{
    /* Enable the Global Interrupts */
    INTCONbits.GIE = 1;
    /* Enable the Peripheral Interrupts */
    INTCONbits.PEIE = 1;
}
```

The following functions are part of the I²C driver. Their implementation can be found below, at the GitHub link.

```
static uint8_t I2C1_open(void);
static void I2C1_close(void);
static void I2C1_startCondition(void);
static void I2C1_stopCondition(void);
static uint8_t I2C1_getAckstatBit(void);
static void I2C1_sendNotAcknowledge(void);
static void I2C1_setReceiveMode(void);
static void I2C1_write1ByteRegister(uint8_t address, uint8_t reg, uint8_t data);
static uint8_t I2C1_read1ByteRegister(uint8_t address, uint8_t reg);
```

The following functions are associated with an I²C transmission state. Their implementation can be found below, at the GitHub link.

```
static void I2C_stateWriteStartComplete(void);
static void I2C_stateWriteAddressSent(void);
static void I2C_stateWriteRegisterSent(void);
static void I2C_stateWriteDataSent(void);
static void I2C_stateReadStart(void);
static void I2C_stateReadStartComplete(void);
static void I2C_stateReadAddressSent(void);
static void I2C_stateReadReceiveEnable(void);
static void I2C_stateReadDataComplete(void);
static void I2C_stateStopComplete(void);
```

The `MSSP1_interruptHandler` function is called every time the SSP1IF flag is triggered. This handler must execute different operations, depending on the current state, which are stored in `I2C1_status.state`.

The `I2C_stateFuncs` vector contains all the function pointers associated with all the I²C transmission states.

The `MSSP1_interruptHandler` function calls the function for the current state, where the state is updated, after which the SSP1IF flag is cleared.

```
static void MSSP1_interruptHandler(void)
{
    /* Call the function associated with the current state */
    I2C_stateFuncs[I2C1_status.state]();

    /* Clear the Interrupt Flag */
    PIR3bits.SSP1IF = 0;
}

void __interrupt() INTERRUPT_InterruptManager (void)
{
    if(INTCONbits.PEIE == 1)
    {
        if(PIE3bits.SSP1IE == 1 && PIR3bits.SSP1IF == 1)
        {
```

```
            MSSP1_interruptHandler();
        }
    }
}
```

The `main` function has multiple responsibilities:

- Initializes the clock frequency, Peripheral Pin Select, ports and I^2C peripheral
- Enables the peripheral and global interrupts
- Sets the slave's I/O Direction (IODIR) register to '0', the value for digital output pins
- Continuously sets the extended port to the value `data` with a write operation in the slave's General Purpose I/O PORT Register; reads the value from the same register and inverts the read value

```
#define I2C_SLAVE_ADDRESS           0x20
#define MCP23008_REG_ADDR_IODIR     0x00
#define MCP23008_REG_ADDR_GPIO      0x09
#define MCP23008_DATA               0x0F
#define PINS_DIGITAL_OUTPUT         0x00

void main(void)
{
    CLK_Initialize();
    PPS_Initialize();
    PORT_Initialize();
    I2C1_Initialize();
    INTERRUPT_Initialize();

    /* Set the initial state to Idle */
    I2C1_status.state = I2C_IDLE;
    /* Set data to use in the I2C operations */
    uint8_t data = MCP23008_DATA;
    /* Set the extended pins as digital output */
    I2C1_write(I2C_SLAVE_ADDRESS, MCP23008_REG_ADDR_IODIR, PINS_DIGITAL_OUTPUT);

    while (1)
    {
        /* Write data to the GPIO port */
        I2C1_write(I2C_SLAVE_ADDRESS, MCP23008_REG_ADDR_GPIO, data);
        /* Read data from the GPIO port */
        data = I2C1_read(I2C_SLAVE_ADDRESS, MCP23008_REG_ADDR_GPIO);
        /* Overwrite data with the inverted data read */
        data = ~data;

        __delay_ms(500);
    }
}
```

View the PIC18F47Q10 Code Example on GitHub
Click to browse repositories

4. References

1. How to install MCC
2. PIC1000: Getting Started with Writing C-Code for PIC16 and PIC18 Technical Brief
3. MCP23008 - 8-Bit I/O Expander with Serial Interface
4. Master Synchronous Serial Port (MSSP) to the Stand-Alone I^2C Module Migration
5. TB3191 - I2C Master Mode

5. Revision History

Document Revision	Date	Comments
A	08/2020	Initial document release

The Microchip Website

Microchip provides online support via our website at www.microchip.com/. This website is used to make files and information easily available to customers. Some of the content available includes:

- **Product Support** – Data sheets and errata, application notes and sample programs, design resources, user's guides and hardware support documents, latest software releases and archived software
- **General Technical Support** – Frequently Asked Questions (FAQs), technical support requests, online discussion groups, Microchip design partner program member listing
- **Business of Microchip** – Product selector and ordering guides, latest Microchip press releases, listing of seminars and events, listings of Microchip sales offices, distributors and factory representatives

Product Change Notification Service

Microchip's product change notification service helps keep customers current on Microchip products. Subscribers will receive email notification whenever there are changes, updates, revisions or errata related to a specified product family or development tool of interest.

To register, go to www.microchip.com/pcn and follow the registration instructions.

Customer Support

Users of Microchip products can receive assistance through several channels:

- Distributor or Representative
- Local Sales Office
- Embedded Solutions Engineer (ESE)
- Technical Support

Customers should contact their distributor, representative or ESE for support. Local sales offices are also available to help customers. A listing of sales offices and locations is included in this document.

Technical support is available through the website at: www.microchip.com/support

Microchip Devices Code Protection Feature

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.

Legal Notice

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, Adaptec, AnyRate, AVR, AVR logo, AVR Freaks, BesTime, BitCloud, chipKIT, chipKIT logo, CryptoMemory, CryptoRF, dsPIC, FlashFlex, flexPWR, HELDO, IGLOO, JukeBlox, KeeLoq, Kleer, LANCheck, LinkMD, maXStylus, maXTouch, MediaLB, megaAVR, Microsemi, Microsemi logo, MOST, MOST logo, MPLAB, OptoLyzer, PackeTime, PIC, picoPower, PICSTART, PIC32 logo, PolarFire, Prochip Designer, QTouch, SAM-BA, SenGenuity, SpyNIC, SST, SST Logo, SuperFlash, Symmetricom, SyncServer, Tachyon, TempTrackr, TimeSource, tinyAVR, UNI/O, Vectron, and XMEGA are registered trademarks of Microchip Technology Incorporated in the U.S.A. and other countries.

APT, ClockWorks, The Embedded Control Solutions Company, EtherSynch, FlashTec, Hyper Speed Control, HyperLight Load, IntelliMOS, Libero, motorBench, mTouch, Powermite 3, Precision Edge, ProASIC, ProASIC Plus, ProASIC Plus logo, Quiet-Wire, SmartFusion, SyncWorld, Temux, TimeCesium, TimeHub, TimePictra, TimeProvider, Vite, WinPath, and ZL are registered trademarks of Microchip Technology Incorporated in the U.S.A.

Adjacent Key Suppression, AKS, Analog-for-the-Digital Age, Any Capacitor, AnyIn, AnyOut, BlueSky, BodyCom, CodeGuard, CryptoAuthentication, CryptoAutomotive, CryptoCompanion, CryptoController, dsPICDEM, dsPICDEM.net, Dynamic Average Matching, DAM, ECAN, EtherGREEN, In-Circuit Serial Programming, ICSP, INICnet, Inter-Chip Connectivity, JitterBlocker, KleerNet, KleerNet logo, memBrain, Mindi, MiWi, MPASM, MPF, MPLAB Certified logo, MPLIB, MPLINK, MultiTRAK, NetDetach, Omniscient Code Generation, PICDEM, PICDEM.net, PICkit, PICtail, PowerSmart, PureSilicon, QMatrix, REAL ICE, Ripple Blocker, SAM-ICE, Serial Quad I/O, SMART-I.S., SQI, SuperSwitcher, SuperSwitcher II, Total Endurance, TSHARC, USBCheck, VariSense, ViewSpan, WiperLock, Wireless DNA, and ZENA are trademarks of Microchip Technology Incorporated in the U.S.A. and other countries.

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

The Adaptec logo, Frequency on Demand, Silicon Storage Technology, and Symmcom are registered trademarks 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.

ISBN: 978-1-5224-6526-3

Quality Management System

For information regarding Microchip's Quality Management Systems, please visit www.microchip.com/quality.

Worldwide Sales and Service

AMERICAS	ASIA/PACIFIC	ASIA/PACIFIC	EUROPE
Corporate Office 2355 West Chandler Blvd. Chandler, AZ 85224-6199 Tel: 480-792-7200 Fax: 480-792-7277 Technical Support: www.microchip.com/support Web Address: www.microchip.com **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 **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 Tel: 317-536-2380 **Los Angeles** Mission Viejo, CA Tel: 949-462-9523 Fax: 949-462-9608 Tel: 951-273-7800 **Raleigh, NC** Tel: 919-844-7510 **New York, NY** Tel: 631-435-6000 **San Jose, CA** Tel: 408-735-9110 Tel: 408-436-4270 **Canada - Toronto** Tel: 905-695-1980 Fax: 905-695-2078	**Australia - Sydney** Tel: 61-2-9868-6733 **China - Beijing** Tel: 86-10-8569-7000 **China - Chengdu** Tel: 86-28-8665-5511 **China - Chongqing** Tel: 86-23-8980-9588 **China - Dongguan** Tel: 86-769-8702-9880 **China - Guangzhou** Tel: 86-20-8755-8029 **China - Hangzhou** Tel: 86-571-8792-8115 **China - Hong Kong SAR** Tel: 852-2943-5100 **China - Nanjing** Tel: 86-25-8473-2460 **China - Qingdao** Tel: 86-532-8502-7355 **China - Shanghai** Tel: 86-21-3326-8000 **China - Shenyang** Tel: 86-24-2334-2829 **China - Shenzhen** Tel: 86-755-8864-2200 **China - Suzhou** Tel: 86-186-6233-1526 **China - Wuhan** Tel: 86-27-5980-5300 **China - Xian** Tel: 86-29-8833-7252 **China - Xiamen** Tel: 86-592-2388138 **China - Zhuhai** Tel: 86-756-3210040	**India - Bangalore** Tel: 91-80-3090-4444 **India - New Delhi** Tel: 91-11-4160-8631 **India - Pune** Tel: 91-20-4121-0141 **Japan - Osaka** Tel: 81-6-6152-7160 **Japan - Tokyo** Tel: 81-3-6880- 3770 **Korea - Daegu** Tel: 82-53-744-4301 **Korea - Seoul** Tel: 82-2-554-7200 **Malaysia - Kuala Lumpur** Tel: 60-3-7651-7906 **Malaysia - Penang** Tel: 60-4-227-8870 **Philippines - Manila** Tel: 63-2-634-9065 **Singapore** Tel: 65-6334-8870 **Taiwan - Hsin Chu** Tel: 886-3-577-8366 **Taiwan - Kaohsiung** Tel: 886-7-213-7830 **Taiwan - Taipei** Tel: 886-2-2508-8600 **Thailand - Bangkok** Tel: 66-2-694-1351 **Vietnam - Ho Chi Minh** Tel: 84-28-5448-2100	**Austria - Wels** Tel: 43-7242-2244-39 Fax: 43-7242-2244-393 **Denmark - Copenhagen** Tel: 45-4485-5910 Fax: 45-4485-2829 **Finland - Espoo** Tel: 358-9-4520-820 **France - Paris** Tel: 33-1-69-53-63-20 Fax: 33-1-69-30-90-79 **Germany - Garching** Tel: 49-8931-9700 **Germany - Haan** Tel: 49-2129-3766400 **Germany - Heilbronn** Tel: 49-7131-72400 **Germany - Karlsruhe** Tel: 49-721-625370 **Germany - Munich** Tel: 49-89-627-144-0 Fax: 49-89-627-144-44 **Germany - Rosenheim** Tel: 49-8031-354-560 **Israel - Ra'anana** Tel: 972-9-744-7705 **Italy - Milan** Tel: 39-0331-742611 Fax: 39-0331-466781 **Italy - Padova** Tel: 39-049-7625286 **Netherlands - Drunen** Tel: 31-416-690399 Fax: 31-416-690340 **Norway - Trondheim** Tel: 47-72884388 **Poland - Warsaw** Tel: 48-22-3325737 **Romania - Bucharest** Tel: 40-21-407-87-50 **Spain - Madrid** Tel: 34-91-708-08-90 Fax: 34-91-708-08-91 **Sweden - Gothenberg** Tel: 46-31-704-60-40 **Sweden - Stockholm** Tel: 46-8-5090-4654 **UK - Wokingham** Tel: 44-118-921-5800 Fax: 44-118-921-5820