
Getting Started with SPI Using MSSP on PIC18

Introduction

Author: Iustinian Bujor, Microchip Technology Inc.

The approach in implementing the SPI 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 Serial Peripheral Interface (SPI) peripheral.

Both peripherals are serial interfaces useful for communicating with other peripherals 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), which allows the advantage of implementing both communication protocols with the same hardware. The dedicated SPI peripheral works similarly to the MSSP, and has more features, such as Receive Only and Transmit Only modes, Double Buffering and Receive and Transmit FIFO.

This technical brief provides information about MSSP on the PIC18-K40 and PIC18-Q10 product families and intends to familiarize the user with PIC® microcontrollers.

The document describes the application area, the modes of operation and the hardware and software requirements of the MSSP module configured in SPI mode.

Throughout the document, the configuration of the peripheral will be described in detail, starting with the location of the SPI pins, the direction of the pins, how to initialize the device as a master or a slave and how to exchange data inside the system. This document covers the following use cases:

- **Sending Data as a Master SPI Device with Multiple Slaves:**
This example shows how to configure the device as a master to control two slave devices and to send data using the polling method.
- **Receiving Data as a Slave SPI Device:**
This example shows how to configure the device as a slave that will wait for the incoming data using the polling method.
- **Exchanging Data as a Slave SPI Device Using Interrupts:**
This example shows how to configure the device as a slave that will wait for the incoming data. In this case, the data transmission/reception will be triggered by interrupts.
- **Changing Data Transfer Type:**
This example shows how to configure the device as a master that will send data with respect to the clock polarity and the clock edge.

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. Also, MCC provides a graphical interface that eases the peripheral configuration process and helps the users to evaluate the peripherals they are not familiar with. 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 tends to be more device specific, 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 QFN40.



View Code Examples on GitHub

Click to browse repositories

Table of Contents

Introduction.....	1
1. Peripheral Overview	4
2. Sending Data as a Master SPI Device with Multiple Slaves.....	6
2.1. MCC Generated Code.....	6
2.2. Foundation Services Generated Code.....	7
2.3. Bare Metal Code.....	8
3. Receiving Data as a Slave SPI Device.....	12
3.1. MCC Generated Code.....	12
3.2. Foundation Services Generated Code.....	13
3.3. Bare Metal Code.....	14
4. Exchanging Data as a Slave SPI Device Using Interrupts.....	17
4.1. MCC Generated Code.....	17
4.2. Foundation Services Generated Code.....	18
4.3. Bare Metal Code.....	20
5. Changing Data Transfer Type.....	23
5.1. MCC Generated Code.....	23
5.2. Foundation Services Generated Code.....	24
5.3. Bare Metal Code.....	25
6. References.....	29
7. Revision History.....	30
The Microchip Website.....	31
Product Change Notification Service.....	31
Customer Support.....	31
Microchip Devices Code Protection Feature.....	31
Legal Notice.....	32
Trademarks.....	32
Quality Management System.....	33
Worldwide Sales and Service.....	34

1. Peripheral Overview

The SPI bus is a synchronous serial data communication bus that operates in Full Duplex mode, using a channel for transmitting and another channel for receiving data. The devices communicate in a master-slave environment, with a single master at a time and one or more slaves.

The SPI bus consists of four signal connections:

- SCK: Serial Clock (output from master)
- SDO: Serial Data Out (data output)
- SDI: Serial Data Input (data input)
- SS: Slave Select (active-low, output from master)

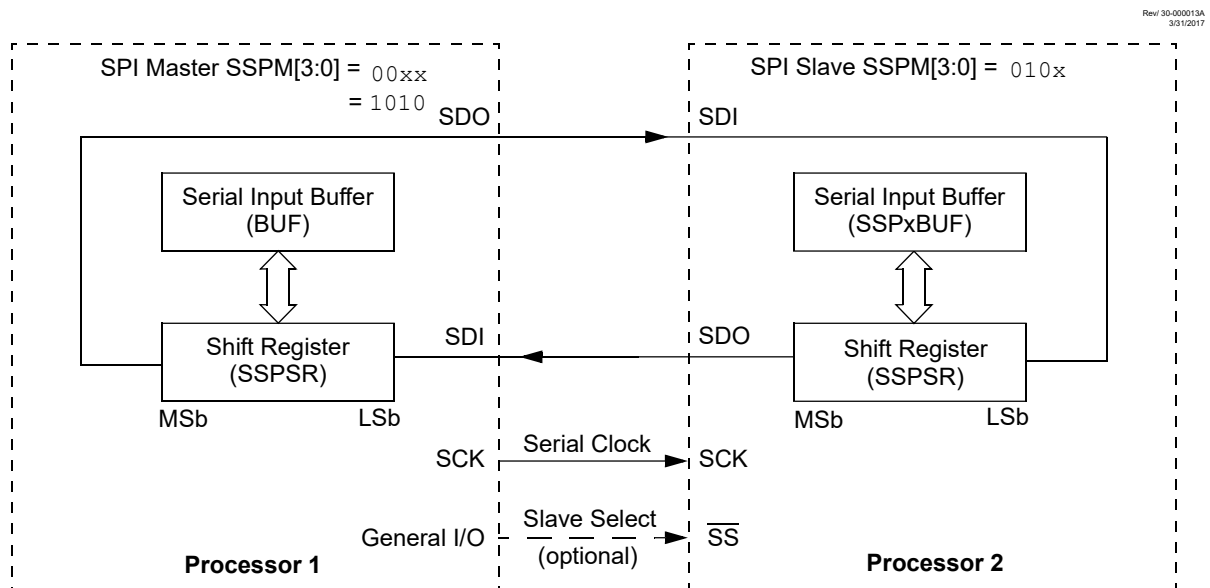
The master device is the only one that can generate a clock. Therefore, it is typically the initiator of the data exchange, although there are methods of slave initiation. The SPI master device uses the same SCK, SDO and SDI channels for all the slaves, but usually individual lines of SS for each of the slaves. However, the daisy-chain feature offers the possibility of using only one SS line to control more than one slave device. The master device selects the desired slave by pulling the SS signal low.

The data to be sent will be stored in the buffer register (SSPxBUF). The master device transmits information out on its SDO output pin, which is connected to and received by the slave's SDI input pin. The slave device transmits information out on its SDO output pin, which is connected to and received by the master's SDI input pin.

Transmissions involve two eight bit-sized shift registers, one in the master and the other in the slave. With either the master or the slave device, data is always shifted out one bit at a time, on the programmed clock edge and with the Most Significant bit (MSb) shifted out first. At the same time, a new Least Significant bit (LSb) is shifted into the same register.

Any write to the SSPxBUF register during transmission/reception of data will be ignored and the Write Collision Detect (WCOL) bit from the MSSP Control 1 (SSPxCON1) register will be set. User software must clear the WCOL bit to allow the following write(s) to the SSPxBUF register to complete successfully. The MSSP Shift register (SSPSR) is not directly readable or writable and can only be accessed by addressing the SSPxBUF register.

Figure 1-1. Typical SPI Connection



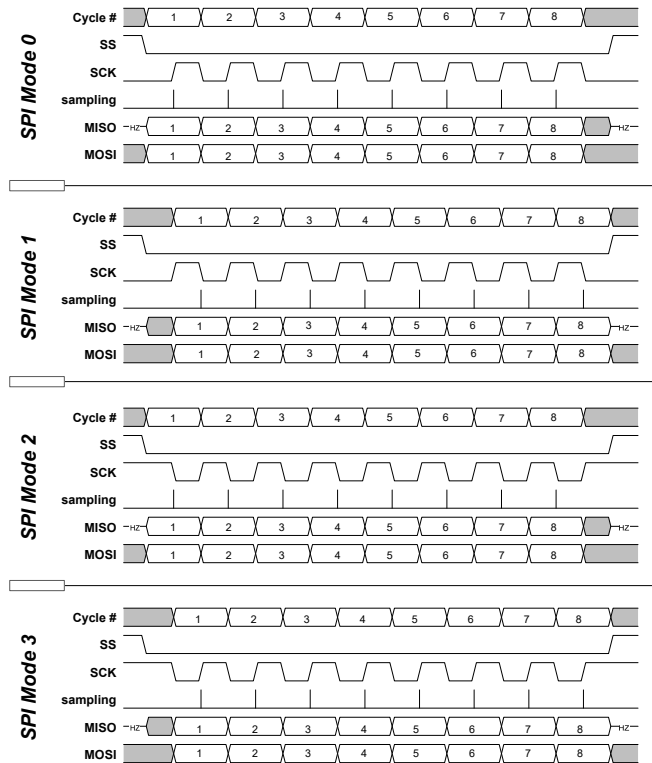
The data transfer type represents the way in which data are transmitted with respect to the clock generation. The clock polarity and the clock edge are the important parameters for data modes.

The clock polarity refers to the level of the signal in Idle state. The signal can be either low in Idle state, and start with a rising edge when transmitting data, or high in Idle state and start with a falling edge when transmitting data.

Depending on the edge, the data are transmitted with respect to the clock on the channel and, therefore, either on a rising or falling edge.

For a better understanding, see the following figure:

Figure 1-2. SPI Data Modes



2. Sending Data as a Master SPI Device with Multiple Slaves

The master is the device that decides when to trigger communication and which slave is the recipient of the message. SPI master devices are generally used in high-speed communication and the focus is to exchange data with other devices acting as slaves (e.g., sensors, memories, or other microcontrollers).

This use case presents how to configure the SPI as a master device along with its pins to send data to two slave devices, one at a time.

To achieve the functionality described by the use case, the following actions will have to be performed:

- System clock initialization
- SPI1 initialization
- Peripheral Pin Selection (PPS) initialization
- Port initialization
- Slave control functions
- Data exchange function

2.1 MCC Generated Code

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 the MCC from the toolbar. Information about 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 **WWDTC** tab, make sure **WDT Disabled** is selected.
 - In the **Programming** tab, make sure **Low-Voltage Programming Enable** is checked.
4. From the Device Resources window, add MSSP1 and do the following configuration:
 - Serial Protocol: SPI
 - Mode: Master
 - SPI Mode: SPI Mode 0
 - Input Data Sampled At: Middle
 - Clock Source Selection: $F_{OSC}/4_SSPxADD$
 - SPI Clock Frequency box: 8000000
5. Open *Pin Manager* → *Grid View* window, select **UQFN40** in the Package field and do the following pin configurations:
 - Set Port C pin 6 (RC6) as output for Slave Select 1 (SS1)
 - Set Port C pin 7 (RC7) as output for Slave Select 2 (SS2)

The SCK, SDO and SDI pins appear alongside the MSSP1 peripheral and have their direction preset.

Figure 2-1. Pin Mapping

Package:	UQFN40		Pin No:		17	18	19	20	21	22	29	28	8	9	10	11	12	13	14	15	30	31	32	33	38	39	40	1	34	35	36	37	2	3	4	5	23	24	25	16
			Port A ▼							Port B ▼							Port C ▼							Port D ▼							Port E ▼									
Module	Function	Direction	0	1	2	3	4	5	6	7	0	1	2	3	4	5	6	7	0	1	2	3	4	5	6	7	0	1	2	3	4	5	6	7	0	1	2	3		
MSSP1 ▼	SCK1	in/out																																						
	SDI1	input																																						
	SDO1	output																																						
OSC	CLKOUT	output																																						
Pin Module ▼	GPIO	input																																						
	GPIO	output																																						

6. Click **Pin Module** in the **Project Resources** tab and set custom names for the SS pins:

- Rename RC6 to Slave1
 - Rename RC7 to Slave2
7. Click **Generate** in the **Project Resources** tab.
 8. In the `main.c` file which has been generated by MCC, change or add the following code:
 - Control of slave devices
 - Data transmission

```
uint8_t writeData = 1;           /* Data that will be transmitted */
uint8_t receiveData;           /* Data that will be received */

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

    while (1)
    {
        SPI1_Open(SPI1_DEFAULT);
        Slave1_SetLow();
        receiveData = SPI1_ExchangeByte(writeData);
        Slave1_SetHigh();
        SPI1_Close();

        SPI1_Open(SPI1_DEFAULT);
        Slave2_SetLow();
        receiveData = SPI1_ExchangeByte(writeData);
        Slave2_SetHigh();
        SPI1_Close();
    }
}
```



View the PIC18F47Q10 Code Example on GitHub

[Click to browse repositories](#)

2.2 Foundation Services Generated Code

To generate this project using Foundation Services Library, follow the next steps:

1. Create a new MPLAB X IDE project for PIC18F47Q10.
2. Open the MCC from the toolbar. 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 **WWDTC** tab, make sure **WDT Disabled** is selected.
 - In the **Programming** tab, make sure **Low-Voltage Programming Enable** is checked.
4. From the *Device Resources* → *Foundation Services* window, add SPIMASTER and do the following configuration:
 - Name: MASTER0
 - SPI Mode: MODE0
 - SPI Data Input Sample: MIDDLE
 - Speed (kHz): 8000
 - SPI: MSSP1
5. Open *Pin Manager* → *Grid View* window, select **UQFN40** in the Package field and do the following pin configurations:
 - Set Port C pin 6 (RC6) as output for Slave Select 1 (SS1)

- Set Port C pin 7 (RC7) as output for Slave Select 2 (SS2)

The SCK, SDO and SDI pins appear alongside the MSSP1 peripheral and have their direction preset.

Figure 2-2. Pin Mapping

Package:	UQFN40	Pin No:	17	18	19	20	21	22	29	28	8	9	10	11	12	13	14	15	30	31	32	33	38	39	40	1	34	35	36	37	2	3	4	5	23	24	25	16	
			Port A ▼								Port B ▼								Port C ▼							Port D ▼							Port E ▼						
Module	Function	Direction	0	1	2	3	4	5	6	7	0	1	2	3	4	5	6	7	0	1	2	3	4	5	6	7	0	1	2	3	4	5	6	7	0	1	2	3	
MSSP1 ▼	SCK1	in/out									⚡	⚡	⚡	⚡	⚡	⚡	⚡	⚡	⚡	⚡	⚡	⚡	⚡	⚡	⚡														
	SDI1	input									⚡	⚡	⚡	⚡	⚡	⚡	⚡	⚡	⚡	⚡	⚡	⚡	⚡	⚡	⚡														
	SDO1	output									⚡	⚡	⚡	⚡	⚡	⚡	⚡	⚡	⚡	⚡	⚡	⚡	⚡	⚡	⚡														
OSC	CLKOUT	output							⚡																														
Pin Module ▼	GPIO	input	⚡	⚡	⚡	⚡	⚡	⚡	⚡	⚡	⚡	⚡	⚡	⚡	⚡	⚡	⚡	⚡	⚡	⚡	⚡	⚡	⚡	⚡	⚡	⚡	⚡	⚡	⚡	⚡	⚡	⚡	⚡	⚡	⚡	⚡	⚡	⚡	
	GPIO	output	⚡	⚡	⚡	⚡	⚡	⚡	⚡	⚡	⚡	⚡	⚡	⚡	⚡	⚡	⚡	⚡	⚡	⚡	⚡	⚡	⚡	⚡	⚡	⚡	⚡	⚡	⚡	⚡	⚡	⚡	⚡	⚡	⚡	⚡	⚡	⚡	

- Click **Pin Module** in the **Project Resources** tab and set custom names for the SS pins:
 - Rename RC6 to Slave1
 - Rename RC7 to Slave2
- Click **Generate** in the **Project Resources** tab.
- In the `main.c` file that has been generated using Foundation Services Library, add the following code:
 - Control of slave devices
 - Data transmission

```
uint8_t writeData = 1;          /* Data that will be transmitted */
uint8_t receiveData;           /* Data that will be received */

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

    while (1)
    {
        spi_master_open(MASTER0);
        Slave1_SetLow();
        receiveData = SPI1_ExchangeByte(writeData);
        Slave1_SetHigh();
        SPI1_Close();

        spi_master_open(MASTER0);
        Slave2_SetLow();
        receiveData = SPI1_ExchangeByte(writeData);
        Slave2_SetHigh();
        SPI1_Close();
    }
}
```



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 Low-Voltage Programming.

```
#pragma config WDTE = OFF
#pragma config LVP = ON
```

The internal oscillator has to be set to the desired value. This example uses the HFINTOSC with a frequency of 64 MHz. This translates into the following function:

```
static void CLK_Initialize(void)
{
    OSCCON1bits.NOSC = 6;      /* HFINTOSC Oscillator */
    OSCFRQbits.HFFRQ = 8;      /* HFFRQ 64 MHz */
}
```

The SPI clock frequency is derived from the main clock of the microcontroller (OSCFRQ) and is reduced using a prescaler or a divider circuit present in the MSSP hardware.

Configure the MSSP in SPI Master mode, with the previous selected clock source. An 8 MHz frequency will result in configuring the device in SPI Master mode with SPI clock = $F_{OSC} / (4 * (SSP1ADD + 1))$. This translates into the following code:

```
static void SPI1_Initialize(void)
{
    SSP1ADD = 0x01;      /* SSP1ADD = 1 */
    SSP1CON1 = 0x2A;      /* Enable module, SPI Master Mode */
}
```

Configuring the location of the pins is independent of the application purpose and the SPI mode. Each microcontroller has its own default physical pin position for peripherals, but the pin positions can be changed using the Peripheral Pin Select (PPS).

The SPI pins can be relocated using the SSPxCLKPPS, SSPxDATPPS and SSPxSSPPS registers for the input channels. Use the RxyPPS registers for the output channels.

The PPS configuration values can be found in the *Peripheral Pin Select Module* section of a device data sheet. For SPI1 in Master mode, only the SDI pin requires input and use it with its default location RC4. SCK was mapped to RC3 and SDO was mapped to RC5. This translates into the following code:

```
static void PPS_Initialize(void)
{
    RC3PPS = 0x0F;      /* SCK channel on RC3 */
    SSP1DATPPS = 0x14;      /* SDI channel on RC4 */
    RC5PPS = 0x10;      /* SDO channel on RC5 */
}
```

Since the master sends data to two slave devices, two SS pins are needed (SS1 and SS2) in this example. For both of them, a General Purpose Input/Output (GPIO) pin was used (RC6 for SS1 and RC7 for SS2).

Table 2-1. SPI Pin Locations

Channel	Pin
SCK	RC3
SDI	RC4
SDO	RC5
SS1	RC6
SS2	RC7

Because the master devices control and initiate transmissions, the SDO, SCK and SS pins must be configured as output while the SDI channel will keep its default direction as input. The following example is based on the relocation of the SPI1 pins made above:

```
static void PORT_Initialize(void)
{
    /* SDI as input; SCK, SDO, SS1, SS2 as output */
}
```

```

    TRISC = 0x17;

    /* SCK, SDI, SDO, SS1, SS2 as digital pins */
    ANSEL = 0x07;
}

```

A master will control a slave by pulling the SS pin low. If the slave has set the direction of its SDO pin to output when the SS pin is low, the SPI driver of the slave will take control of the SDI pin of the master. This will shift data out from its Transmit Buffer register.

All slave devices can receive a message, but only those with the SS pin pulled low can send data back. It is not recommended to enable more than one slave in a typical connection since all of them will try to respond to the message. With the master having only one SDI channel, the transmission will result in a write collision.

Before sending data, the user must pull one of the configured SS signals low to let the correspondent slave device know it is the recipient of the message.

```

static void SPI1_slave1Select(void)
{
    LATCbits.LATC6 = 0;          /* Set SS1 pin value to LOW */
}

```

Once the user writes new data into the Buffer register, the hardware starts a new transfer. This will generate the clock on the line and shift out the bits. The bits are shifted out starting with the Most Significant bit (MSb).

When the hardware finishes shifting all the bits, it sets the Buffer Full Status bit. The user must check the state of the flag before writing new data into the register by constantly reading the value of the bit (polling). If polling is not done, a write collision will occur.

```

static uint8_t SPI1_exchangeByte(uint8_t data)
{
    SSP1BUF = data;

    while(!PIR3bits.SSP1IF)      /* Wait until data is exchanged */
    {
        ;
    }
    PIR3bits.SSP1IF = 0;

    return SSP1BUF;
}

```

The user can pull the SS channel high if there is nothing left to transmit.

```

static void SPI1_slave1Deselect(void)
{
    LATCbits.LATC6 = 1;          /* Set SS1 pin value to HIGH */
}

```

The selection of the slave devices and the data transmission are done in the *main* function.

```

int main(void)
{
    CLK_Initialize();
    PPS_Initialize();
    PORT_Initialize();
    SPI1_Initialize();

    while(1)
    {
        SPI1_slave1Select();
        receiveData = SPI1_exchangeByte(writeData);
        SPI1_slave1Deselect();

        SPI1_slave2Select();
        receiveData = SPI1_exchangeByte(writeData);
        SPI1_slave2Deselect();
    }
}

```



View the PIC18F47Q10 Code Example on GitHub

[Click to browse repositories](#)

3. Receiving Data as a Slave SPI Device

The slave devices are represented by actuators, sensors, external memories, display drivers and more. Slaves usually do not initiate any action; they only act whenever the master initiates. Although there are cases when slave devices can initiate a transmission, their behavior in such a scenario is not presented in this use case. A slave must always be available and has to wait until the master pulls its SS channel low.

This use case presents how to configure the SPI as a slave device, along with its pins to receive data from a master device.

To achieve the functionality described by the use case, the following actions will have to be performed:

- System clock initialization
- SPI1 initialization
- PPS initialization
- Port initialization
- Polling check
- Data exchange function

3.1 MCC Generated Code

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. 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, make sure **WDT Disabled** is selected.
 - In the **Programming** tab, make sure **Low-Voltage Programming Enable** is checked.
4. From the Device Resources window, add MSSP1 and do the following configuration:
 - Serial Protocol: SPI
 - Mode: Slave
 - SPI Mode: SPI Mode 0
 - Enable Slave Select: checked
5. Open *Pin Manager* → *Grid View* window and select **UQFN40** in the Package field. The SCK, SDO, SDI and SS pins appear alongside the MSSP1 peripheral and have their direction preset.

Figure 3-1. Pin Mapping

Package:	UQFN40		Pin No:		17	18	19	20	21	22	29	28	8	9	10	11	12	13	14	15	30	31	32	33	38	39	40	1	34	35	36	37	2	3	4	5	23	24	25	16
			Port A ▼							Port B ▼							Port C ▼							Port D ▼							Port E ▼									
Module	Function	Direction	0	1	2	3	4	5	6	7	0	1	2	3	4	5	6	7	0	1	2	3	4	5	6	7	0	1	2	3	4	5	6	7	0	1	2	3		
MSSP1 ▼	SCK1	input																																						
	SDI1	input																																						
	SDO1	output																																						
	SS1	input																																						
OSC	CLKOUT	output																																						
Pin Module ▼	GPIO	input																																						
	GPIO	output																																						

6. Click **Generate** in the **Project Resources** tab.
7. In the `main.c` file which has been generated by MCC, change or add the following code:
 - Peripheral initialization

- Data receiving

```
uint8_t receiveData;      /* Data that will be received */
uint8_t writeData = 1;    /* Data that will be transmitted */

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

    SPI1_Open(SPI1_DEFAULT);

    while (1)
    {
        if(!RA5_GetValue()) /* SS line is LOW */
        {
            receiveData = SPI1_ExchangeByte(writeData);
        }
    }
}
```



View the PIC18F47Q10 Code Example on GitHub

[Click to browse repositories](#)

3.2 Foundation Services Generated Code

To generate this project using Foundation Services Library, follow the next steps:

1. Create a new MPLAB X IDE project for PIC18F47Q10.
2. Open MCC from the toolbar. Information about 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 **WWDTC** tab, make sure **WDT Disabled** is selected.
 - In the **Programming** tab, make sure **Low-Voltage Programming Enable** is checked.
4. From the *Device Resources* → *Foundation Services* window, add SPISLAVE and do the following configuration:
 - Name: SLAVE0
 - SPI Mode: MODE0
 - SPI: MSSP1
5. Go to *Project Resources* → *System* → *Interrupt Module* → *Easy Setup* and uncheck the box right next to Pin Module.
6. Open *Pin Manager* → *Grid View* window, select **UQFN40** in the Package field and do the following pin configuration:
 - Set Port A pin 5 (RA5) as input for Slave Select (SS)

The SCK, SDO and SDI pins appear alongside the MSSP1 peripheral and have their direction preset.

Figure 3-2. Pin Mapping

Package:	UQFN40	Pin No:	17	18	19	20	21	22	29	28	8	9	10	11	12	13	14	15	30	31	32	33	38	39	40	1	34	35	36	37	2	3	4	5	23	24	25	16				
			Port A ▼								Port B ▼								Port C ▼								Port D ▼								Port E ▼							
Module	Function	Direction	0	1	2	3	4	5	6	7	0	1	2	3	4	5	6	7	0	1	2	3	4	5	6	7	0	1	2	3	4	5	6	7	0	1	2	3				
MSSP1 ▼	SCK1	in/out																																								
	SDI1	input																																								
	SDO1	output																																								
OSC	CLKOUT	output																																								
Pin Module ▼	GPIO	input																																								
	GPIO	output																																								
RESET	MCLR	input																																								

7. Click **Generate** in the **Project Resources** tab.
8. In the `main.c` file which has been generated using Foundation Services Library, add the following code:
 - Peripheral initialization
 - Data receiving

```

uint8_t receiveData;          /* Data that will be received */
uint8_t writeData = 1;        /* Data that will be transmitted */

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

    spil_open(SLAVE0_CONFIG);

    while (1)
    {
        if(!SS_GetValue())    /* SS line is LOW */
        {
            receiveData = spil_exchangeByte(writeData);
        }
    }
}

```



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 Low-Voltage Programming.

```

#pragma config WDTE = OFF
#pragma config LVP = ON

```

The internal oscillator has to be set to the desired value. This example uses the HFINTOSC with a frequency of 64 MHz, which translates into the following function:

```

static void CLK_Initialize(void)
{
    OSCCON1bits.NOSC = 6;    /* HFINTOSC Oscillator */

    OSCFRQbits.HFFRQ = 8;    /* HFFRQ 64 MHz */
}

```

Since the slave gets its clock signal from the master device, there is no point in changing the clock divider of the peripheral. This change will have no effect in SPI Slave mode. However, the hardware peripheral has to sample the

data received on their SDI channel. For the data signal to be correctly reconstructed, the main clock frequency of the device must be at least double the frequency received on the SPI SCK channel.

Next is an example on how to configure the MSSP in SPI Slave mode.

```
static void SPI1_Initialize(void)
{
    /* Enable module, SPI Slave Mode */
    SSP1CON1 = 0x24;
}
```

Configuring the location of the pins is independent of the application purpose and the SPI mode. Each microcontroller has its own default physical pin position for peripherals, but the pin positions can be changed using the Peripheral Pin Select (PPS).

The SPI pins can be relocated using the SSPxCLKPPS, SSPxDATPPS and SSPxSSPPS registers for the input channels. Use the RxyPPS registers for the output channels.

The PPS configuration values can be found in the *Peripheral Pin Select Module* section of a device data sheet. For SPI1 in Slave mode, the SDI and SS pins require input, so use them with their default locations (RC4 for SDI and RA5 for SS). SCK was mapped to RC3 and SDO was mapped to RC5 in this example. This translates into the following code:

```
static void PPS_Initialize(void)
{
    SSP1SSPPS = 0x05;          /* SS channel on RA5 */
    RC3PPS = 0x0F;             /* SCK channel on RC3 */
    SSP1DATPPS = 0x14;         /* SDI channel on RC4 */
    RC5PPS = 0x10;             /* SDO channel on RC5 */
}
```

Table 3-1. SPI Pin Locations

Channel	Pin
SS	RA5
SCK	RC3
SDI	RC4
SDO	RC5

Since the slave device receives the incoming transmissions, the SDI, SCK and SS will keep their default direction as input while the SDO channel must be configured as output. The following example is based on the relocation of the SPI1 pins made above:

```
static void PORT_Initialize(void)
{
    /* SDO as output; SDI, SCK as input */
    TRISC = 0xDF;

    /* SS as digital pin */
    ANSELA = 0xDF;

    /* SCK, SDI, SDO as digital pins */
    ANSELC = 0xC7;
}
```

All the slave devices connected to the SPI bus will receive the message sent on their SDI channel by the master device. A slave cannot respond to a message unless the SS channel is pulled low. When the master device pulls the SS pin low, the SPI peripheral of the slave device will take control of its SDO pin and will shift data out.

Checking the value of SS is done through the polling method.

```
if(!PORTAbits.RA5)           /* SS line is LOW */
```

The peripheral will signal the reception of a new data by activating the SSPxIF flag of the PIR3 register. The user has to check the value of the bit also through polling.

After the received data was processed as wanted, the flag needs to be cleared. This is done by writing a '1' in the bit location.

```
static uint8_t SPI1_exchangeByte(uint8_t data)
{
    SSP1BUF = data;

    while(!PIR3bits.SSP1IF) /* Wait until data is exchanged */
    {
        ;
    }
    PIR3bits.SSP1IF = 0;

    return SSP1BUF;
}
```

The checking of the SS line and the data receiving is done in the *main* function.

```
int main(void)
{
    CLK_Initialize();
    PPS_Initialize();
    PORT_Initialize();
    SPI1_Initialize();

    while(1)
    {
        if(!PORTAbits.RA5)           /* SS line is LOW */
        {
            receiveData = SPI1_exchangeByte(writeData);
        }
    }
}
```



View the PIC18F47Q10 Code Example on GitHub

[Click to browse repositories](#)

4. Exchanging Data as a Slave SPI Device Using Interrupts

The slave devices are represented by actuators, sensors, external memories, display drivers and more. Slaves usually do not initiate any action; they only act whenever the master initiates. Although there are cases when slave devices can initiate a transmission, their behavior in this scenario is not presented in this use case.

This use case presents how to configure the SPI as a slave device, along with its pins to exchange data with a master device. The data received will be interrupt driven.

To achieve the functionality described by the use case, the following actions will have to be performed:

- System clock initialization
- SPI1 initialization
- PPS initialization
- Port initialization
- Interrupts initialization
- Data exchange function
- SPI1 interrupt handling

4.1 MCC Generated Code

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. Information about 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, make sure **WDT Disabled** is selected.
 - In the **Programming** tab, make sure **Low-Voltage Programming Enable** is checked.
4. From the Device Resources window, add MSSP1 and do the following configuration:
 - **Hardware Settings** tab
 - Serial Protocol: SPI
 - Mode: Slave
 - SPI Mode: SPI Mode 0
 - Enable Slave Select: checked
 - **Interrupt Settings** tab
 - Enable SPI Interrupt: checked
5. Open *Pin Manager* → *Grid View* window and select **UQFN40** in the Package field. The SCK, SDO, SDI and SS pins appear alongside the MSSP1 peripheral and have their direction preset.

Figure 4-1. Pin Mapping

Package:	UQFN40		Pin No:	17	18	19	20	21	22	29	28	8	9	10	11	12	13	14	15	30	31	32	33	38	39	40	1	34	35	36	37	2	3	4	5	23	24	25	16
			Port A ▼							Port B ▼							Port C ▼							Port D ▼							Port E ▼								
Module	Function	Direction	0	1	2	3	4	5	6	7	0	1	2	3	4	5	6	7	0	1	2	3	4	5	6	7	0	1	2	3	4	5	6	7	0	1	2	3	
MSSP1 ▼	SCK1	input																																					
	SDI1	input																																					
	SDO1	output																																					
	SS1	input																																					
OSC	CLKOUT	output																																					
Pin Module ▼	GPIO	input																																					
	GPIO	output																																					

6. Click **Generate** in the **Project Resources** tab.
7. In the `main.c` file which has been generated by MCC, change or add the following code:

- Enable the global and peripheral interrupts
- Add the MSSP1 interrupt function
- Set the MSSP1 interrupt handler initialize

```
static void MSSP1_interruptHandler(void) ;

volatile uint8_t receiveData;      /* Data that will be received */
volatile uint8_t writeData = 1;    /* Data that will be transmitted */

static void MSSP1_interruptHandler(void)
{
    receiveData = SPI1_ExchangeByte(writeData) ;
}

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

    SPI1_Open(SPI1_DEFAULT) ;

    SPI1_SetInterruptHandler(MSSP1_interruptHandler) ;

    // Enable the Global Interrupts
    INTERRUPT_GlobalInterruptEnable() ;

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

    // Enable the Peripheral Interrupts
    INTERRUPT_PeripheralInterruptEnable() ;

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

    while (1)
    {
        ;
    }
}
```



View the PIC18F47Q10 Code Example on GitHub

[Click to browse repositories](#)

4.2 Foundation Services Generated Code

To generate this project using Foundation Services Library, follow the next steps:

1. Create a new MPLAB X IDE project for PIC18F47Q10.
2. Open MCC from the toolbar. Information about 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, make sure **WDT Disabled** is selected.
 - In the **Programming** tab, make sure **Low-Voltage Programming Enable** is checked.
4. From the *Device Resources* → *Foundation Services* window, add SPISLAVE and do the following configuration:
 - Name: SLAVE0
 - SPI Mode: MODE0

Exchanging Data as a Slave SPI Device Using ...

- SPI: MSSP1
- 5. Go to *Project Resources* → *System* → *Interrupt Module* → *Easy Setup*. Uncheck the box right next to Pin Module and check the box right next to MSSP1-SSPI.
- 6. Open *Pin Manager* → *Grid View* window, select **UQFN40** in the Package field and do the following pin configuration:
 - Set Port A pin 5 (RA5) as input for Slave Select (SS)

The SCK, SDO and SDI pins appear alongside the MSSP1 peripheral and have their direction preset

Figure 4-2. Pin Mapping

Package:	UQFN40	Pin No:	17	18	19	20	21	22	29	28	8	9	10	11	12	13	14	15	30	31	32	33	38	39	40	1	34	35	36	37	2	3	4	5	23	24	25	16
			Port A ▼							Port B ▼							Port C ▼							Port D ▼							Port E ▼							
Module	Function	Direction	0	1	2	3	4	5	6	7	0	1	2	3	4	5	6	7	0	1	2	3	4	5	6	7	0	1	2	3	4	5	6	7	0	1	2	3
MSSP1 ▼	SCK1	in/out																																				
	SDI1	input																																				
	SDO1	output																																				
OSC	CLKOUT	output																																				
Pin Module ▼	GPIO	input																																				
	GPIO	output																																				
RESET	MCLR	input																																				

7. Click **Generate** in the **Project Resources** tab.
8. In the `main.c` file which has been generated using Foundation Services Library, add the following code:
 - Enable the global and peripheral interrupts
 - Add the MSSP1 interrupt function
 - Set the MSSP1 interrupt handler initialize

```
static void MSSP1_interruptHandler(void) ;

volatile uint8_t receiveData;      /* Data that will be received */
volatile uint8_t writeData = 1;    /* Data that will be transmitted */

static void MSSP1_interruptHandler(void)
{
    receiveData = spi1_exchangeByte(writeData);
}

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

    spi_slave_init();

    spi1_setSpiISR(MSSP1_interruptHandler);

    // Enable the Global Interrupts
    INTERRUPT_GlobalInterruptEnable();

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

    // Enable the Peripheral Interrupts
    INTERRUPT_PeripheralInterruptEnable();

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

    while (1)
    {
        ;
    }
}
```



View the PIC18F47Q10 Code Example on GitHub
Click to browse repositories

4.3 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 Low-Voltage Programming.

```
#pragma config WDTE = OFF
#pragma config LVP = ON
```

The internal oscillator has to be set to the desired value. This example uses the HFINTOSC with a frequency of 64 MHz, which translates into the following function:

```
static void CLK_Initialize(void)
{
    OSCCON1bits.NOSC = 6;          /* HFINTOSC Oscillator */
    OSCFRCbits.HFRC = 8;          /* HFRC 64 MHz */
}
```

Since the slave gets its clock signal from the master device, there is no point in changing the clock divider of the peripheral. This change will have no effect in SPI Slave mode. However, the hardware peripheral has to sample the data received on their SDI channel. For the data signal to be correctly reconstructed, the main clock frequency of the device must be at least double the clock frequency received on the SPI SCK channel.

Configure the MSSP in SPI Slave mode. To receive data using interrupts, the Interrupt Enable bit (SSPxIE) of the PIE3 register must be set.

```
static void SPI1_Initialize(void)
{
    /* Enable module, SPI Slave Mode */
    SSP1CON1 = 0x24;

    /* Enable MSSP interrupts */
    PIE3bits.SSP1IE = 1;
}
```

This allows the SPI module to trigger an interrupt every time it has received data on its SDI pin.

Configuring the location of the pins is independent of the application purpose and the SPI mode. Each microcontroller has its own default physical pin position for peripherals, but the pin positions can be changed using the Peripheral Pin Select (PPS).

The SPI pins can be relocated by using the SSPxCLKPPS, SSPxDATPPS and SSPxSSPPS registers for the input channels. Use the RxyPPS registers for the output channels.

The PPS configuration values can be found in the *Peripheral Pin Select Module* section of a device data sheet. For SPI1 in Slave mode, the SDI and SS pins require input, so use them with their default location (RC4 for SDI and RA5 for SS). SCK was mapped to RC3 and SDO was mapped to RC5 in this example. This translates into the following code:

```
static void PPS_Initialize(void)
{
    SSP1SSPPS = 0x05;          /* SS channel on RA5 */
    RC3PPS = 0x0F;            /* SCK channel on RC3 */
    SSP1DATPPS = 0x14;         /* SDI channel on RC4 */
}
```

```

    RC5PPS = 0x10;          /* SDO channel on RC5 */
}

```

Table 4-1. Pin Selection for SPI Slave

Channel	Pin
SS	RA5
SCK	RC3
SDI	RC4
SDO	RC5

Since the slave device receives the incoming transmissions, the SDI, SCK and SS will keep their default direction as input while the SDO channel must be configured as output. The following example is based on the relocation of the SPI pins made above:

```

static void PORT_Initialize(void)
{
    /* SDO as output; SDI, SCK as input */
    TRISC = 0xDF;

    /* SS as digital pin */
    ANSELA = 0xDF;

    /* SCK, SDI, SDO as digital pins */
    ANSELC = 0xC7;
}

```

Before any transfer of data, the interrupts of the microcontroller must be activated. This is done by setting the Global Interrupt Enable (GIE) and the Peripheral Interrupt Enable (PIE) bits of the INTCON register.

```

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

```

The code snippet below shows how to write to the Buffer register, clear the Interrupt flag and read the received data. The user can choose what to do with the received data and what to write back to the master:

```

static uint8_t SPI1_exchangeByte(uint8_t data)
{
    SSP1BUF = data;

    while(!PIR3bits.SSP1IF) /* Wait until data is exchanged */
    {
        ;
    }
    PIR3bits.SSP1IF = 0;

    return SSP1BUF;
}

```

This exchange function is integrated in the interrupt handler of the MSSP1.

```

static void MSSP1_interruptHandler(void)
{
    receiveData = SPI1_exchangeByte(writeData);
}

```

The interrupt handler is handled by the interrupt manager.

```

void __interrupt() INTERRUPT_InterruptManager(void)
{
    if(INTCONbits.PEIE == 1)

```

```
{  
    if (PIE3bits.BCL1IE == 1 && PIR3bits.BCL1IF == 1)  
    {  
        MSSP1_InterruptHandler();  
    }  
    else if (PIE3bits.SSP1IE == 1 && PIR3bits.SSP1IF == 1)  
    {  
        MSSP1_InterruptHandler();  
    }  
}
```



View the PIC18F47Q10 Code Example on GitHub

[Click to browse repositories](#)

5. Changing Data Transfer Type

This use case presents how to configure the SPI as a master device along with its pins to send data to a slave device in Data Mode 3.

In this mode, the Idle clock state is high and the data are transmitted on a falling edge of the clock signal (from high to low).

Note: Both the master and the slave devices must be configured in the same way so one can decode correctly what the other encoded.

To achieve the functionality described by the use case, the following actions will have to be performed:

- System clock initialization
- SPI1 initialization
- PPS initialization
- Port initialization
- Slave control functions
- Data exchange function

5.1 MCC Generated Code

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. Information about 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 **WWDTC** tab, make sure **WDT Disabled** is selected.
 - In the **Programming** tab, make sure **Low-Voltage Programming Enable** is checked.
4. From the Device Resources window, add MSSP1 and do the following configuration:
 - Serial Protocol: SPI
 - Mode: Master
 - SPI Mode: SPI Mode 3
 - Input Data Sampled At: Middle
 - Clock Source Selection: FOSC/4_SSPxADD
 - SPI Clock Frequency box: 8000000
5. Open *Pin Manager* → *Grid View* window, select **UQFN40** in the Package field and do the following pin configuration:
 - Set Port C pin 6 (RC6) as output for Slave Select (SS)

The SCK, SDO and SDI pins appear alongside the MSSP1 peripheral and have their direction preset.

Figure 5-1. Pin Mapping

Package:	UQFN40	Pin No:	17	18	19	20	21	22	29	28	8	9	10	11	12	13	14	15	30	31	32	33	38	39	40	1	34	35	36	37	2	3	4	5	23	24	25	16
			Port A ▼							Port B ▼							Port C ▼							Port D ▼							Port E ▼							
Module	Function	Direction	0	1	2	3	4	5	6	7	0	1	2	3	4	5	6	7	0	1	2	3	4	5	6	7	0	1	2	3	4	5	6	7	0	1	2	3
MSSP1 ▼	SCK1	in/out																																				
	SDI1	input																																				
	SDO1	output																																				
OSC	CLKOUT	output																																				
Pin Module ▼	GPIO	input																																				
	GPIO	output																																				

6. Click **Pin Module** in the **Project Resources** tab and set the custom name for the SS pin:

- Rename RC6 to Slave
- 7. Click **Generate** in the **Project Resources** tab.
- 8. In the `main.c` file which has been generated by MCC, change or add the following code:
 - Control of slave device
 - Data transmission

```
uint8_t writeData = 1;          /* Data that will be transmitted */
uint8_t receiveData = 0;       /* Data that will be received */

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

    while (1)
    {
        SPI1_Open(SPI1_DEFAULT);
        Slave_SetLow();
        receiveData = SPI1_ExchangeByte(writeData);
        Slave_SetHigh();
        SPI1_Close();
    }
}
```



View the PIC18F47Q10 Code Example on GitHub

[Click to browse repositories](#)

5.2 Foundation Services Generated Code

To generate this project using Foundation Services Library, follow the next steps:

1. Create a new MPLAB X IDE project for PIC18F47Q10.
2. Open the MCC from the toolbar. Information about 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, make sure **WDT Disabled** is selected.
 - In the **Programming** tab, make sure **Low-Voltage Programming Enable** is checked.
4. From the *Device Resources* → *Foundation Services* window, add SPIMASTER and do the following configuration:
 - Name: MASTER0
 - SPI Mode: MODE3
 - SPI Data Input Sample: MIDDLE
 - Speed (kHz): 8000
 - SPI: MSSP1
5. Open *Pin Manager* → *Grid View* window, select **UQFN40** in the Package field and do the following pin configuration:
 - Set Port C pin 6 (RC6) as output for Slave Select (SS)

The SCK, SDO and SDI pins appear alongside the MSSP1 peripheral and have their direction preset.

Figure 5-2. Pin Mapping

Package:	UQFN40	Pin No:	17	18	19	20	21	22	29	28	8	9	10	11	12	13	14	15	30	31	32	33	38	39	40	1	34	35	36	37	2	3	4	5	23	24	25	16	
			Port A ▼							Port B ▼							Port C ▼							Port D ▼							Port E ▼								
Module	Function	Direction	0	1	2	3	4	5	6	7	0	1	2	3	4	5	6	7	0	1	2	3	4	5	6	7	0	1	2	3	4	5	6	7	0	1	2	3	
MSSP1 ▼	SCK1	in/out									⚡	⚡	⚡	⚡	⚡	⚡	⚡	⚡	⚡	⚡	⚡	⚡	⚡	⚡	⚡	⚡													
	SDI1	input									⚡	⚡	⚡	⚡	⚡	⚡	⚡	⚡	⚡	⚡	⚡	⚡	⚡	⚡	⚡	⚡													
	SDO1	output									⚡	⚡	⚡	⚡	⚡	⚡	⚡	⚡	⚡	⚡	⚡	⚡	⚡	⚡	⚡	⚡													
OSC	CLKOUT	output							⚡																														
Pin Module ▼	GPIO	input	⚡	⚡	⚡	⚡	⚡	⚡	⚡	⚡	⚡	⚡	⚡	⚡	⚡	⚡	⚡	⚡	⚡	⚡	⚡	⚡	⚡	⚡	⚡	⚡	⚡	⚡	⚡	⚡	⚡	⚡	⚡	⚡	⚡	⚡	⚡	⚡	
	GPIO	output	⚡	⚡	⚡	⚡	⚡	⚡	⚡	⚡	⚡	⚡	⚡	⚡	⚡	⚡	⚡	⚡	⚡	⚡	⚡	⚡	⚡	⚡	⚡	⚡	⚡	⚡	⚡	⚡	⚡	⚡	⚡	⚡	⚡	⚡	⚡	⚡	

- Click **Pin Module** in the **Project Resources** tab and set the custom name for the SS pin:
 - Rename RC6 to Slave
- Click **Generate** in the **Project Resources** tab.
- In the `main.c` file which has been generated using Foundation Services Library, add the following code:
 - Control of slave device
 - Data transmission

```
uint8_t writeData = 1;          /* Data that will be transmitted */
uint8_t receiveData = 0;        /* Data that will be received */

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

    while (1)
    {
        spi_master_open(MASTERO);
        Slave_SetLow();
        receiveData = SPI1_ExchangeByte(writeData);
        Slave_SetHigh();
        SPI1_Close();
    }
}
```



View the PIC18F47Q10 Code Example on GitHub

Click to browse repositories

5.3 Bare Metal Code

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

The first step is to configure the microcontroller to disable the Watchdog Timer and to enable Low-Voltage Programming.

```
#pragma config WDTE = OFF
#pragma config LVP = ON
```

The internal oscillator has to be set to the desired value. This example uses the HFINTOSC with a frequency of 64 MHz, which translates into the following function:

```
static void CLK_Initialize(void)
{
    OSCCON1bits.NOSC = 6;          /* HFINTOSC Oscillator */
    OSCFRQbits.HFFRQ = 8;          /* HFFRQ 64 MHz */
}
```

The SPI clock frequency is derived from the main clock of the microcontroller and is reduced using a prescaler or divider circuit present in the MSSP hardware.

The clock polarity can be changed by modifying the value of the SCK Release Control bit from the SSPxCON1 register. The bit is cleared by default, so the Idle state of the SCK is at low level. For the next example, the bit will be set, which results in the Idle state of SCK being at high level.

Similarly with clock polarity, the clock edge can be changed by modifying the value of the Clock Select bit from the SSPxSTAT register. The bit is cleared by default, so the transmission occurred on the transition from Idle to Active Clock state.

Configure the MSSP in SPI Master mode, with the previous selected clock source. The master will send data in Data Mode 3. An 8 MHz frequency will result in configuring the device in SPI Master mode with SPI clock = $F_{OSC} / (4 * (SSP1ADD + 1))$.

This translates into the following code:

```
static void SPI1_Initialize(void)
{
    /* SSP1ADD = 1 */
    SSP1ADD = 0x01;

    /* Enable module, MSSP in SPI Master mode, CKP = 1 */
    SSP1CON1 = 0x3A;
}
```

Configuring the location of the pins is independent of the application purpose and the SPI mode. Each microcontroller has its own default physical pin position for peripherals, but the pin positions can be changed using the Peripheral Pin Select (PPS).

The SPI pins can be relocated using the SSPxCLKPPS, SSPxDATPPS and SSPxSSPPS registers for the input channels. Use the RxyPPS registers for output channels.

The PPS configuration values can be found in the *Peripheral Pin Select Module* section of a device data sheet. For SPI1 in Master mode, only the SDI pin requires input and use it with its default location RC4. SCK was mapped to RC3 and SDO was mapped to RC5 in this example. This translates into the following code:

```
static void PPS_Initialize(void)
{
    RC3PPS = 0x0F;           /* SCK channel on RC3 */
    SSP1DATPPS = 0x14;       /* SDI channel on RC4 */
    RC5PPS = 0x10;           /* SDO channel on RC5 */
}
```

This example has the master send to a slave device, so one SS pin is needed. A General Purpose Input/Output (GPIO) pin was used (RC6).

Table 5-1. SPI Pin Locations

Channel	Pin
SCK	RC3
SDI	RC4
SDO	RC5
SS	RC6

Since the master devices control and initiate transmissions, the SDO, SCK and SS pins must be configured as output while the SDI channel will keep its default direction as input. The following example is based on the relocation of the SPI1 pins made above.

```
static void PORT_Initialize(void)
{
    /* SDI as input; SCK, SDO, SS as output */
}
```

```
TRISC = 0x97;

/* SCK, SDI, SDO, SS as digital pins */
ANSELC = 0x87;
}
```

A master will control a slave by pulling the SS pin low. If the slave has set the direction of its SDO pin to output when the SS pin is low, the SPI driver of the slave will take control of the SDI pin of the master. This will shift data out from its Transmit Buffer register.

All slave devices can receive a message, but only those with the SS pin pulled low can send data back. It is not recommended to enable more than one slave in a typical connection since all of them will try to respond to the message. Since the master has only one SDI channel, the transmission will result in a write collision.

Before sending data, the user must pull one of the configured SS signals low to let the correspondent slave device know it is the recipient of the message.

```
static void SPI1_slaveSelect(void)
{
    LATCbits.LATC6 = 0;          /* Set SS1 pin value to LOW */
}
```

Once the user writes new data into the Buffer register, the hardware starts a new transfer. This will generate the clock on the line and shift out the bits. The bits are shifted out starting with the Most Significant bit (MSb).

When the hardware finishes shifting all the bits, it sets the Buffer Full Status bit. The user must check the state of the flag before writing new data into the register by constantly reading the value of the bit (polling). If not done, a write collision will occur.

```
static uint8_t SPI1_exchangeByte(uint8_t data)
{
    SSP1BUF = data;

    while(!PIR3bits.SSP1IF) /* Wait until data is exchanged */
    {
        ;
    }
    PIR3bits.SSP1IF = 0;

    return SSP1BUF;
}
```

The user can pull the SS channel high if there is nothing left to transmit.

```
static void SPI1_slaveDeselect(void)
{
    LATCbits.LATC6 = 1;          /* Set SS1 pin value to HIGH */
}
```

The selection of the slave device and the data transmission are done in the *main* function.

```
int main(void)
{
    CLK_Initialize();
    PPS_Initialize();
    PORT_Initialize();
    SPI1_Initialize();

    while(1)
    {
        SPI1_slaveSelect();
        receiveData = SPI1_exchangeByte(writeData);
        SPI1_slaveDeselect();
    }
}
```



View the PIC18F47Q10 Code Example on GitHub

[Click to browse repositories](#)

6. References

1. [MPLAB Code Configurator User's Guide](#)
2. [PIC1000: Getting Started with Writing C-Code for PIC16 and PIC18](#)
3. [TB3215-Getting Started with SPI](#)

7. Revision History

Doc Rev.	Date	Comments
B	10/2020	Updated PIC1000 link from References section
A	5/2020	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 specifications contained in their particular Microchip Data Sheet.
- Microchip believes that its family of products is secure when used in the intended manner and under normal conditions.
- There are dishonest and possibly illegal methods being used in attempts to breach the code protection features of the Microchip devices. We believe that these methods require using the Microchip products in a manner outside the operating specifications contained in Microchip's Data Sheets. Attempts to breach these code protection features, most likely, cannot be accomplished without violating Microchip's intellectual property rights.
- Microchip is willing to work with any customer who is concerned about the integrity of its code.
- Neither Microchip nor any other semiconductor manufacturer can guarantee the security of its code. Code protection does not mean that we are guaranteeing the product is "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 is provided for the sole purpose of designing with and using Microchip products. Information 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.

THIS INFORMATION IS PROVIDED BY MICROCHIP "AS IS". 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 ANY IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY, AND FITNESS FOR A PARTICULAR PURPOSE OR WARRANTIES RELATED TO ITS CONDITION, QUALITY, OR PERFORMANCE.

IN NO EVENT WILL MICROCHIP BE LIABLE FOR ANY INDIRECT, SPECIAL, PUNITIVE, INCIDENTAL OR CONSEQUENTIAL LOSS, DAMAGE, COST OR EXPENSE OF ANY KIND WHATSOEVER RELATED TO THE INFORMATION OR ITS USE, HOWEVER CAUSED, EVEN IF MICROCHIP HAS BEEN ADVISED OF THE POSSIBILITY OR THE DAMAGES ARE FORESEEABLE. TO THE FULLEST EXTENT ALLOWED BY LAW, MICROCHIP'S TOTAL LIABILITY ON ALL CLAIMS IN ANY WAY RELATED TO THE INFORMATION OR ITS USE WILL NOT EXCEED THE AMOUNT OF FEES, IF ANY, THAT YOU HAVE PAID DIRECTLY TO MICROCHIP FOR THE INFORMATION. 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.

© 2020, Microchip Technology Incorporated, Printed in the U.S.A., All Rights Reserved.

ISBN: 978-1-5224-6877-6

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	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