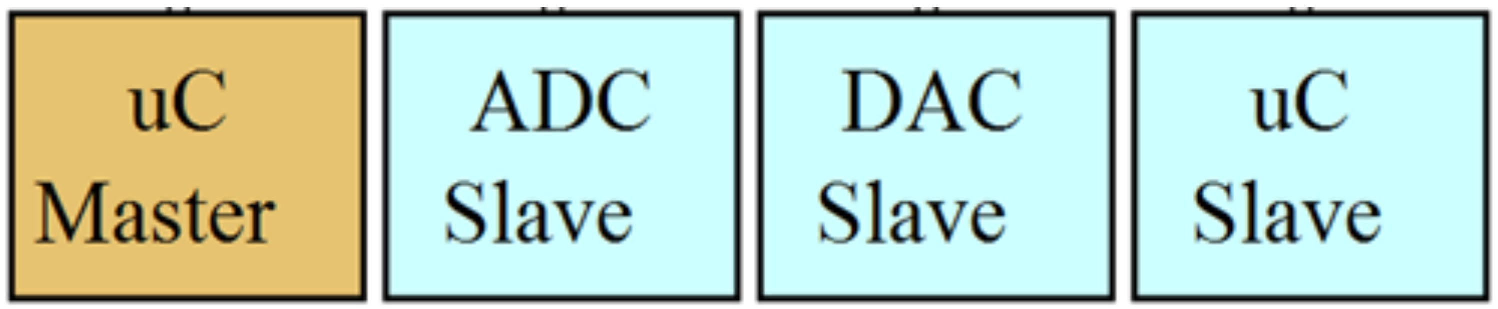
Serial Peripheral Interface (SPI)

ECE 2534

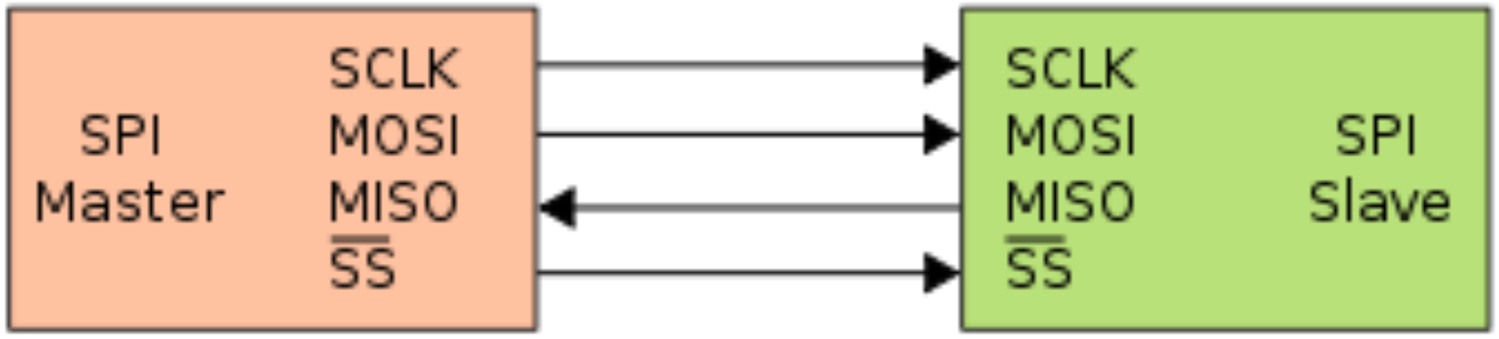
Motivation: a microcontroller needs to communicate with several peripherals (maybe scattered around a circuit board)



Serial Peripheral Interface (SPI)

- **SPI is a standard for full-duplex, synchronous, serial communication**
- Pronounced “spy”
- “Full-duplex”: can send and receive at the same time
- “Synchronous”: a clock signal is provided
- Normally used at the board level
- It was developed by Motorola in the 1980s, and it became a de-facto standard
- Often called a “4-wire” serial bus

SPI example (1 controller, 1 peripheral)

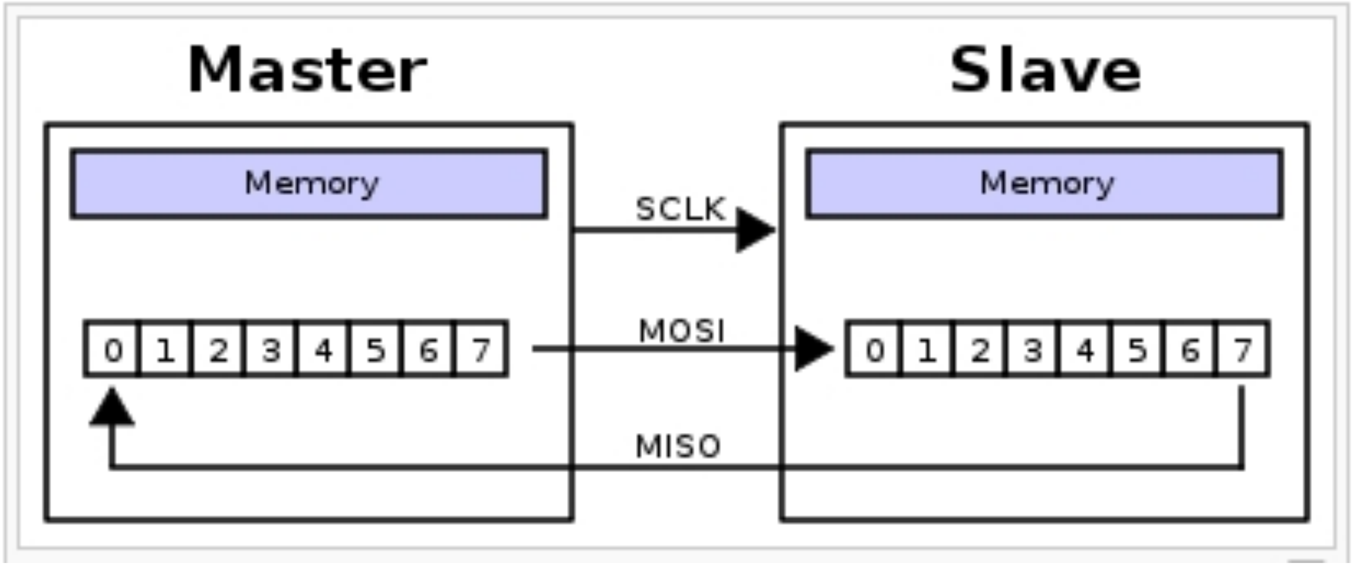


Source: Wikipedia

- SCLK: Serial Clock
- MOSI: Master Out Slave In
- MISO: Master In Slave Out
- SS: Slave Select

- SCLK: Serial Clock
  - generated by the Master
  - synchronizes the transfers
- MOSI: Master Out Slave In
  - carries data from the Master to the Slave
- MISO: Master In Slave Out
  - carries data from the Slave to the Master
- SS: Slave Select
  - generated by the Master to enable a particular peripheral device

SPI example (1 controller, 1 peripheral)

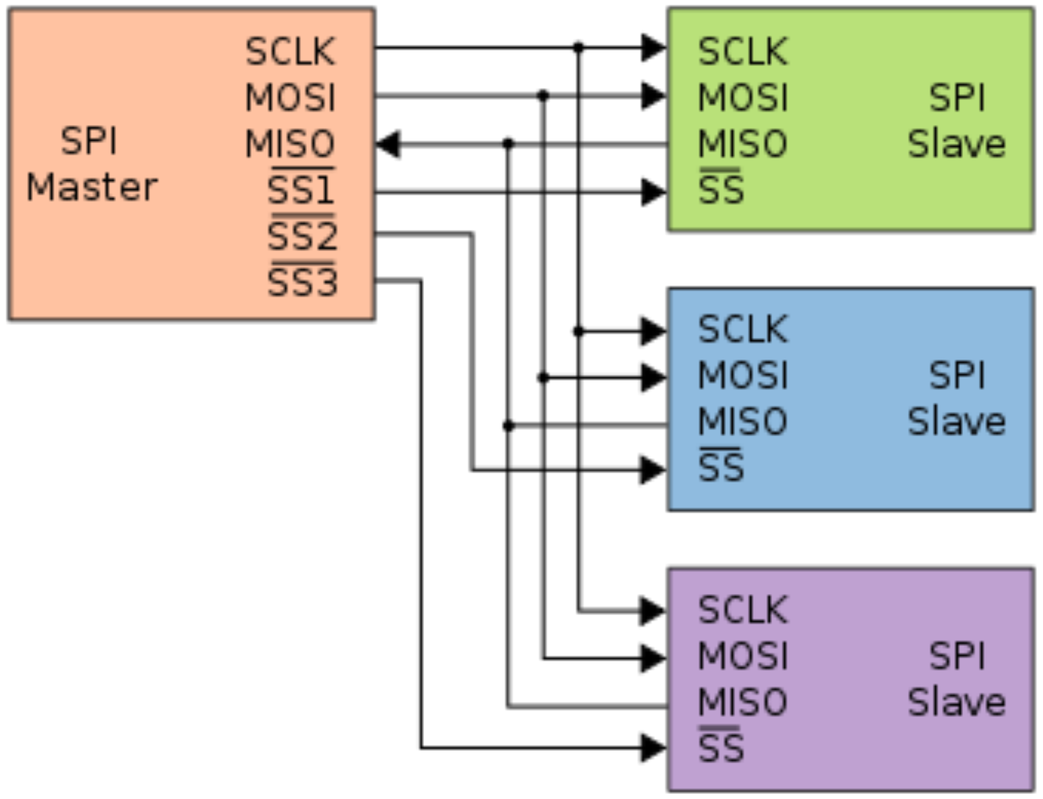


Source: Wikipedia

- Typical arrangement: use 2 shift registers to form an inter-chip circular buffer
- (The SS signal not shown here)

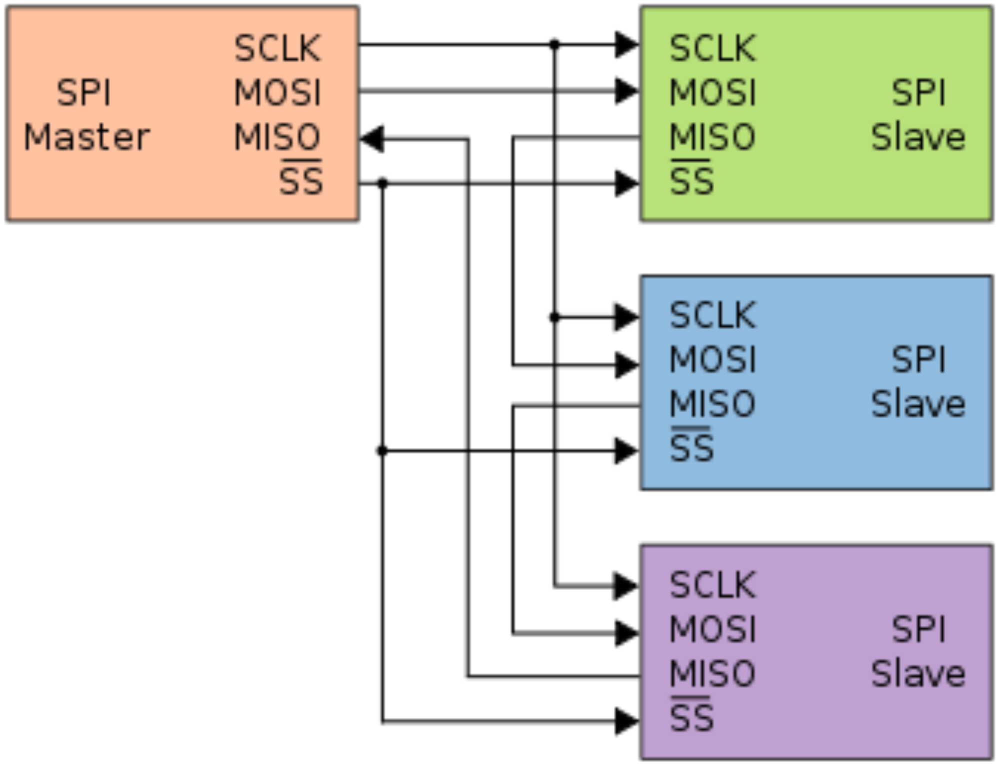
SPI example (1 controller, 3 peripherals)

Independent slave configuration: MISO must be 3-state or Open-Drain



SPI example (1 controller, 3 peripherals)

Daisy-chain configuration



How “standard” is SPI?

- Unlike other communications frameworks such as I^{2}C, no formal standard was ever established for SPI
- Different interpretations and implementations exist
  - Number of bits per transfer is implementation-dependent
  - No standard addressing scheme
  - Maximum data rate is not stated
  - ACK/NACK hand-shaking is not specified

Example SPI communication

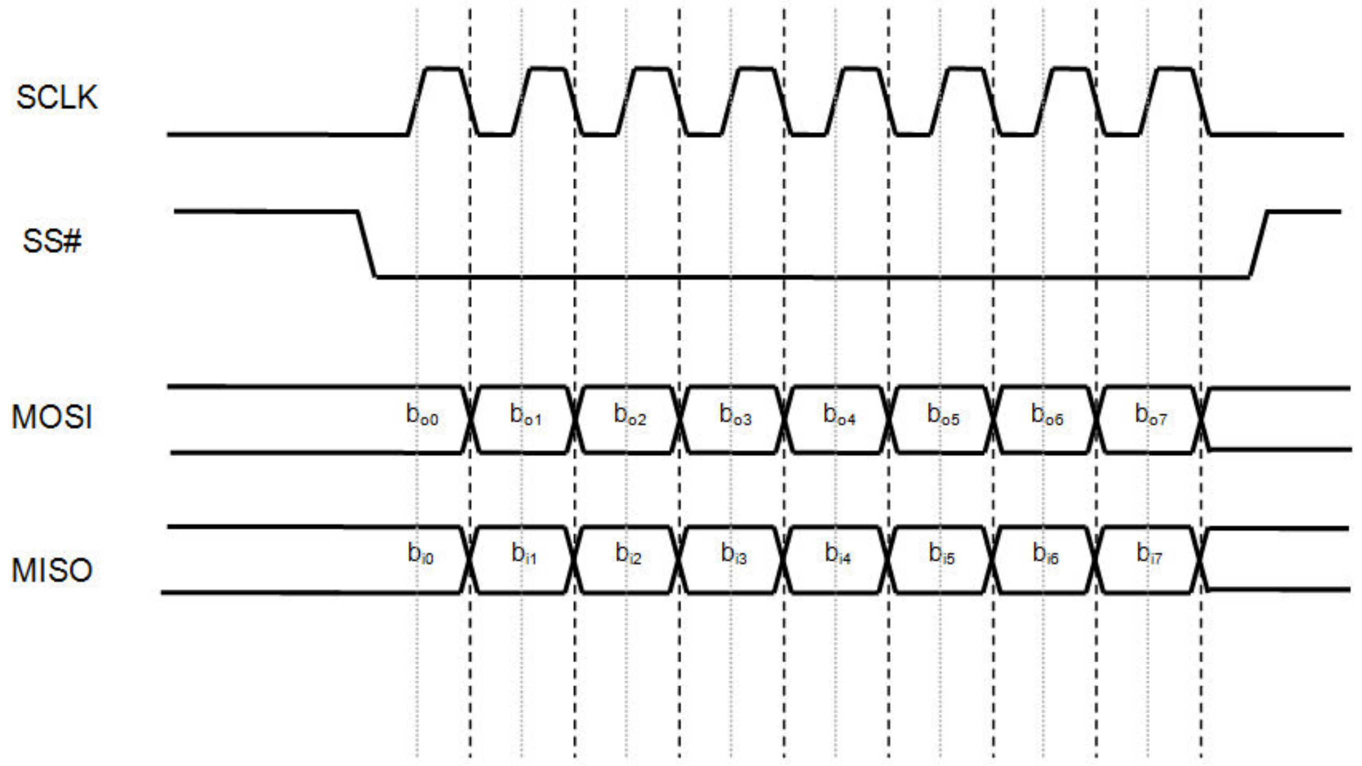
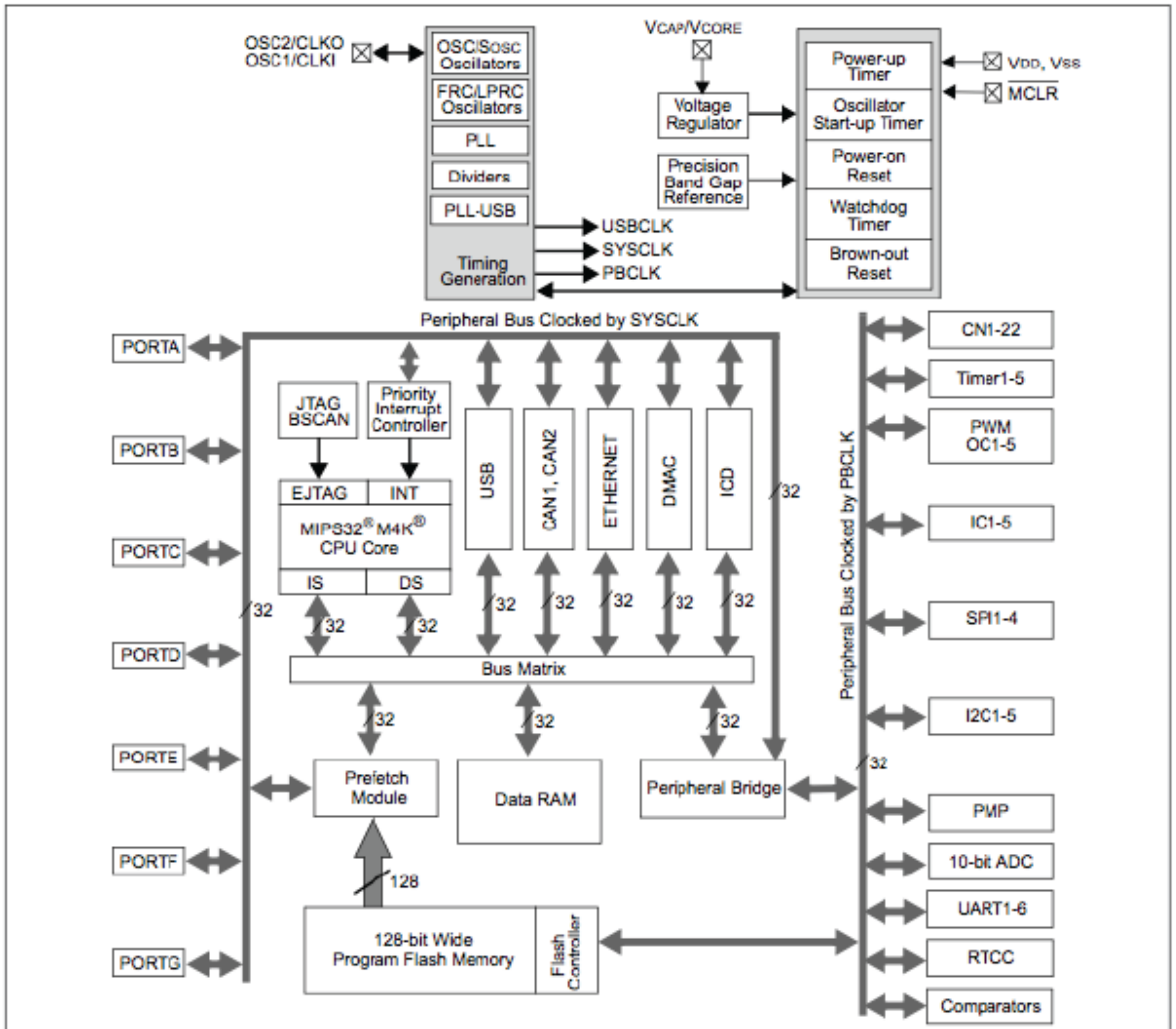


Figure 2 : A simple SPI communication. Data bits on MOSI and MISO toggle on the SCLK falling edge and are sampled on the SCLK rising edge. The SPI mode defines which SCLK edge is used for toggling data and which SCLK edge is used for sampling data.

SPI implementation on the PIC32

FIGURE 1-1: BLOCK DIAGRAM[1,2]



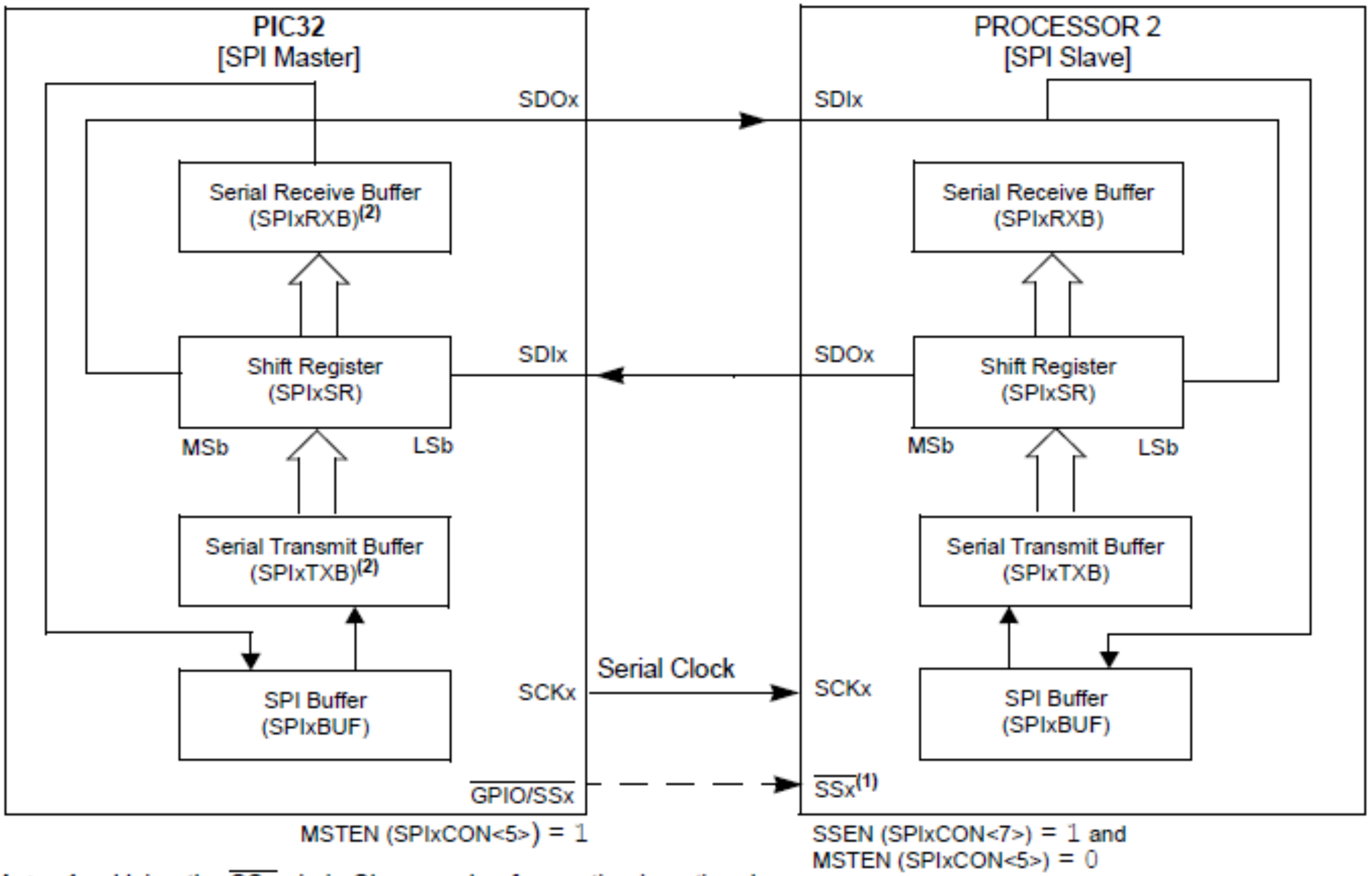
Note 1: Some features are not available on all device variants.

2: BOR functionality is provided when the on-board voltage regulator is enabled.

SPI implementation on the PIC32

- Our PIC32 contains four SPI modules
- Each can act as a master or as a slave
- Three data widths: 8, 16, or 32
- MSB is sent first
- “Normal mode”: One clock pulse per data bit
- “Framed mode”: Clock runs continuously
- “Audio Protocol Interface mode”

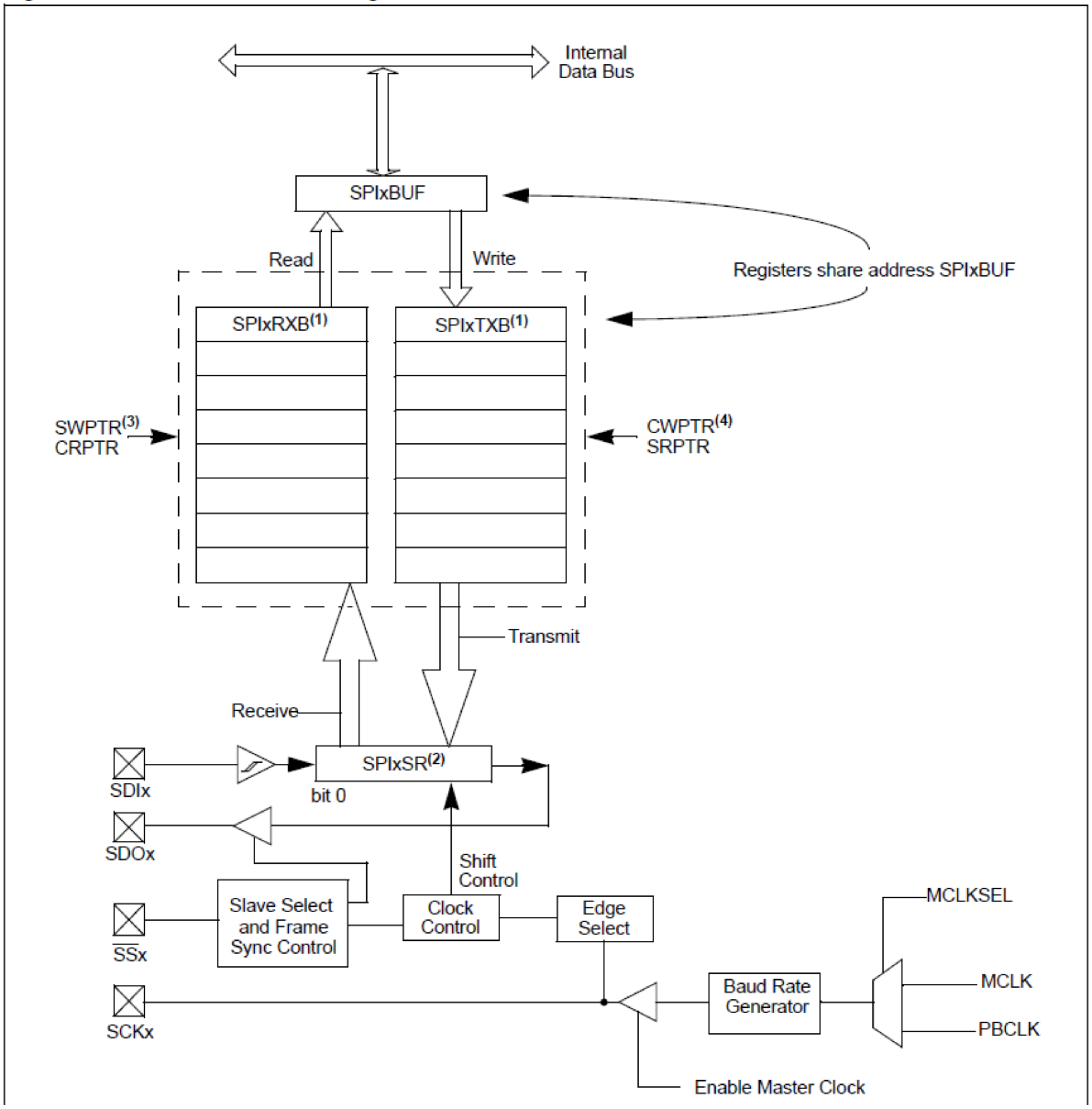
Figure 23-8: SPI Master/Slave Connection Diagram



Note 1: Using the $\overline{\text{SSx}}$ pin in Slave mode of operation is optional.

2: User must write transmit data to SPIxBUF and read received data from SPIxBUF. The SPIxTXB and SPIxRXB registers are memory mapped to SPIxBUF.

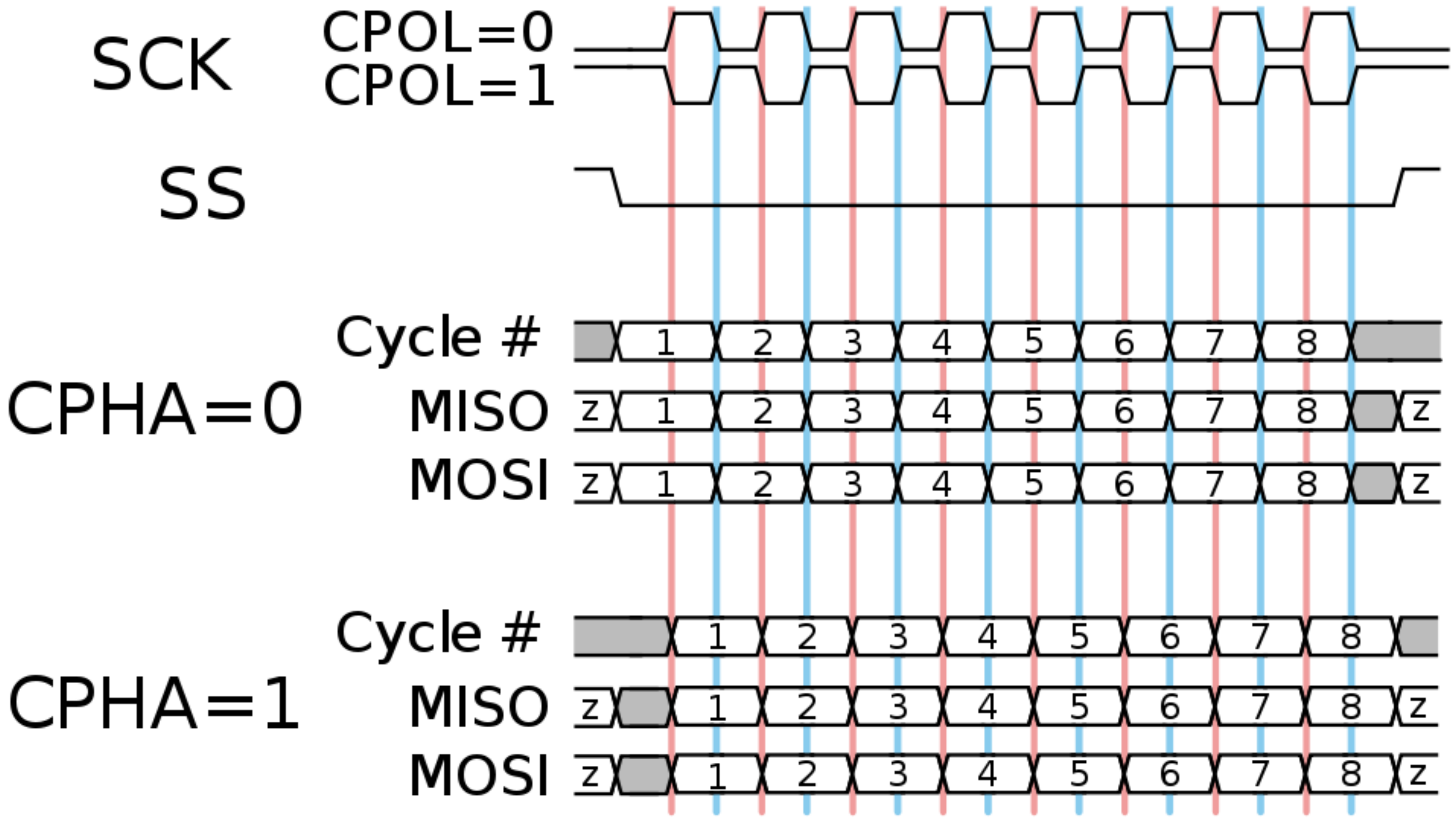
Figure 23-1: SPI Module Block Diagram



Controlling the Timing

- CPOL (*Clock Polarity*) – Determines the base value of the clock. [Called CKP in the PIC32.]
  - 0 – Clock is normally low
  - 1 – Clock is normally high
- CPHA (*Clock Phase*) – Determines the clock edge on which data is sampled. [Called CKE in the PIC32.]
  - 0 – Sample on the leading edge
  - 1 – Sample on the trailing edge

SPI Timing


SCK
CPOL=0
CPOL=1
SS
CPHA=0
Cycle #
MISO
MOSI
CPHA=1
Cycle #
MISO
MOSI
1
2
3
4
5
6
7
8
z

CONTROL AND STATUS REGISTERS

- The SPI module consists of the following Special Function Registers (SFRs):
- **SPIxCON: SPI Control Register**
- **SPIxCON2: SPI Control Register 2**

  These 2 registers enable control of the SPI module's operation.
- **SPIxSTAT: SPI Status Register**

  This register contains status flags indicating the SPI module's state during operation
- **SPIxBUF: SPI Buffer Register**

  Writing or reading this register actually maps to writing to the Transmit register (SPIxTXB) or the Receive register (SPIxRXB)
- **SPIxBRG: SPI Baud Rate Register**
- This read/write register specifies the rate of data transfer

MASTER MODE INITIALIZATION

Perform the following steps to set up the SPI module for the Master mode operation:

1. Disable the SPI interrupts in the respective IECx register.
2. Stop and reset the SPI module by clearing the ON bit.
3. Clear the receive buffer.
4. Clear the ENHBUF bit (SPIxCON<16>) if using Standard Buffer mode or set the bit if using Enhanced Buffer mode.
5. If SPI interrupts are not going to be used, skip this step and continue to step 5. Otherwise the following additional steps are performed:
   a) Clear the SPIx interrupt flags/events in the respective IFSx register.
   b) Write the SPIx interrupt priority and subpriority bits in the respective IPCx register.
   c) Set the SPIx interrupt enable bits in the respective IECx register.
6. Write the Baud Rate register, SPIxBRG.
7. Clear the SPIROV bit (SPIxSTAT<6>).
8. Write the desired settings to the SPIxCON register with MSTEN (SPIxCON<5>) = 1.
9. Enable SPI operation by setting the ON bit (SPIxCON<15>).
10. Write the data to be transmitted to the SPIxBUF register. Transmission (and reception) will start as soon as data is written to the SPIxBUF register.

MASTER MODE OPERATION

The following progression describes the SPI module operation in Master mode:

1. Once the module is set up for Master mode operation and enabled, data to be transmitted is written to SPIxBUF register. The SPITBE bit (SPIxSTAT<3>) is cleared.
2. The contents of SPIxTXB are moved to the shift register, SPIxSR (see Figure 23-8), and the SPITBE bit is set by the module.
3. A series of 8/16/32 clock pulses shifts 8/16/32 bits of transmit data from SPIxSR to the SDOx pin and simultaneously shifts the data at the SDIx pin into SPIxSR.
4. When the transfer is complete, the following events will occur:
   a) The SPIxRXIF interrupt flag bit is set. SPI interrupts can be enabled by setting the SPIxRXIE interrupt enable bit. The SPIxRXIF flag is not cleared automatically by the hardware.
   b) Also, when the ongoing transmit and receive operation is completed, the contents of SPIxSR are moved to SPIxRXB.
   c) The SPIRBF bit (SPIxSTAT<0>) is set by the module, indicating that the receive buffer is full. Once SPIxBUF is read by the user code, the hardware clears the SPIRBF bit. In Enhanced Buffer mode the SPIRBE bit (SPIxSTAT<5>) is set when the SPIxRXB FIFO buffer is completely empty and cleared when not empty.
5. If the SPIRBF bit is set (the receive buffer is full) when the SPI module needs to transfer data from SPIxSR to SPIxRXB, the module will set the SPIROV bit (SPIxSTAT<6>) indicating an overflow condition.
6. Data to be transmitted can be written to SPIxBUF by the user software at any time, if the SPITBE bit (SPIxSTAT<3>) is set. The write can occur while SPIxSR is shifting out the previously written data, allowing continuous transmission. In Enhanced Buffer mode the SPITBF bit (SPIxSTAT<1>) is set when the SPIxTXB FIFO buffer is completely full and clear when it is not full.

SPI PLIB Functions

c:

→ Program Files (x86)

    → Microchip

        → xc32

            → v1.31

                → pic32mx

                    → include

                        → peripheral

                            → spi_5xx_6xx_7xx.h

Some SPI PLIB Functions

-
  ```
  SpiChnOpen(SpiChannel chn,
             SpiOpenFlags oFlags,
             unsigned int srcClkDiv);
  ```

-
  ```
  SpiChnPutC(SpiChannel chn,
             unsigned int data);
  ```

-
  ```
  data = SpiChnGetC(SpiChannel chn);
  ```

SUMMARY

- The **Serial Peripheral Interface** is a popular way to connect several devices at the board level
  - full-duplex
  - synchronous
  - SCLK, MOSI, MISO, SS
  - normally use a separate select line for each peripheral
- A popular alternative:
  the **I²C** (***Inter-Integrated Circuit***) bus