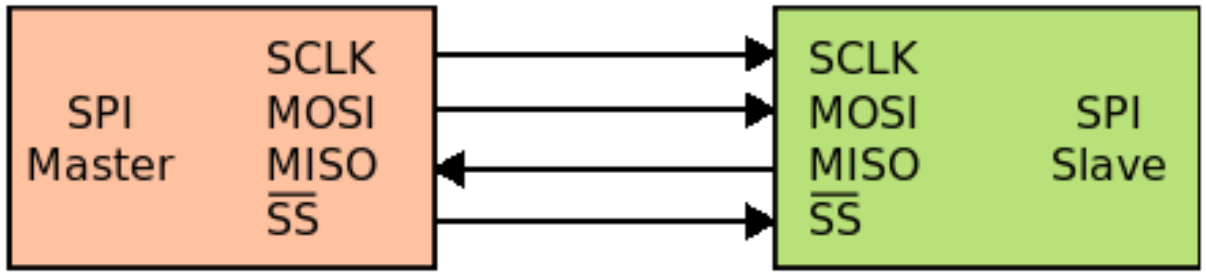
SPI

Serial Peripheral Interface

Allows high-speed synchronous data transfer between the device and peripheral units.

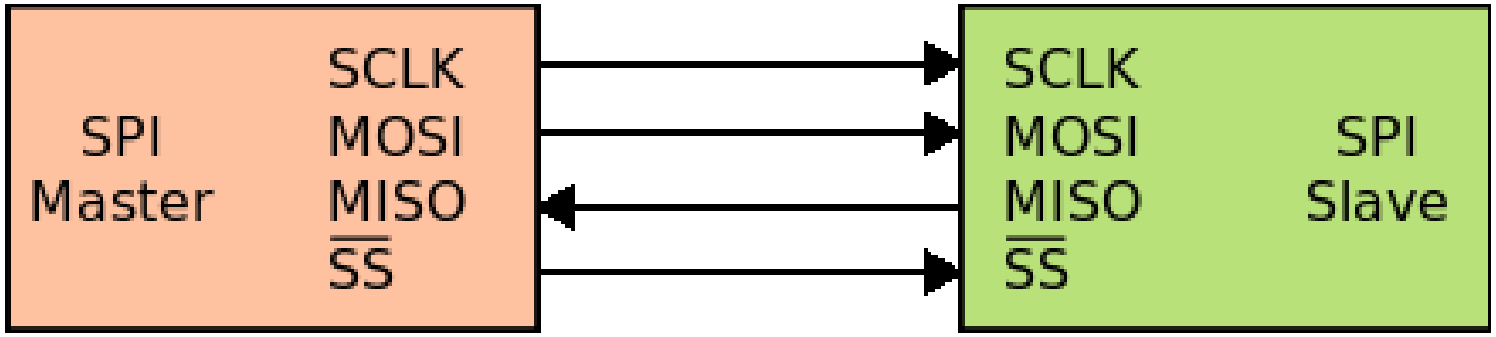
Serial Peripheral Interface Bus

- A synchronous serial data link standard.
- Named by Motorola that operates in full duplex mode.
- Devices communicate in master/slave mode.
- Master device initiates the data frame.
- Multiple slave devices are allowed with individual slave select (chip select) lines.
- Sometimes SPI is called a "four-wire" serial bus.



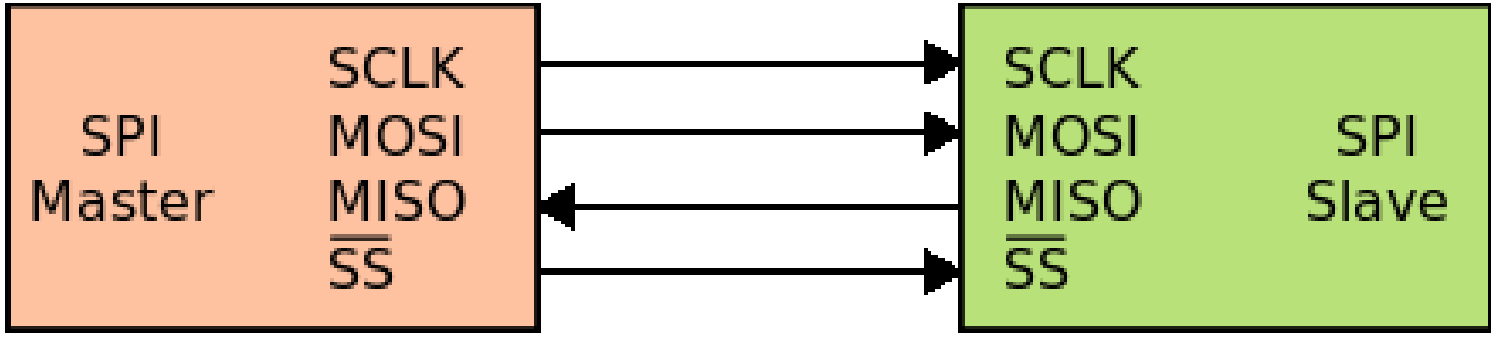
SPI: Interface

- The SPI bus specifies four logic signals:
  - SCLK: Serial Clock (output from master)
  - MOSI; SIMO: Master Output, Slave Input (output from master)
  - MISO; SOMI: Master Input, Slave Output (output from slave)
  - $\overline{SS}$: Slave Select (active low, output from master)

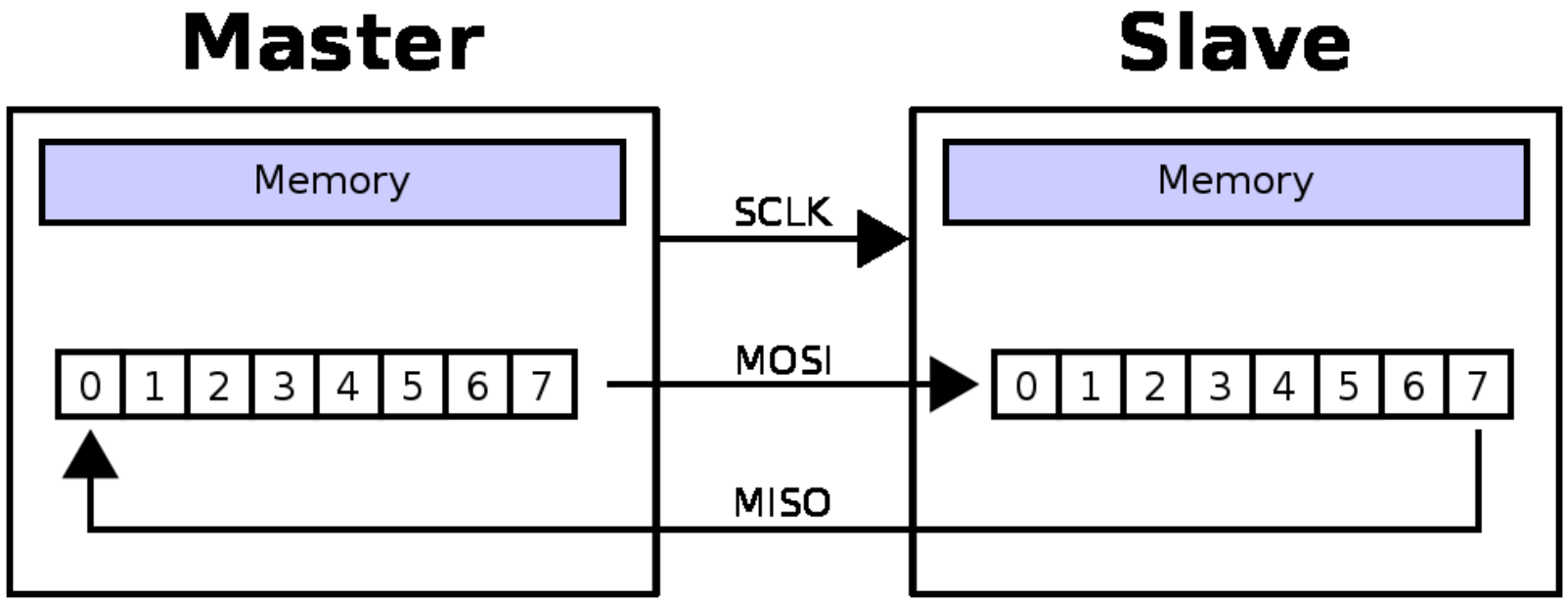


SPI: Interface

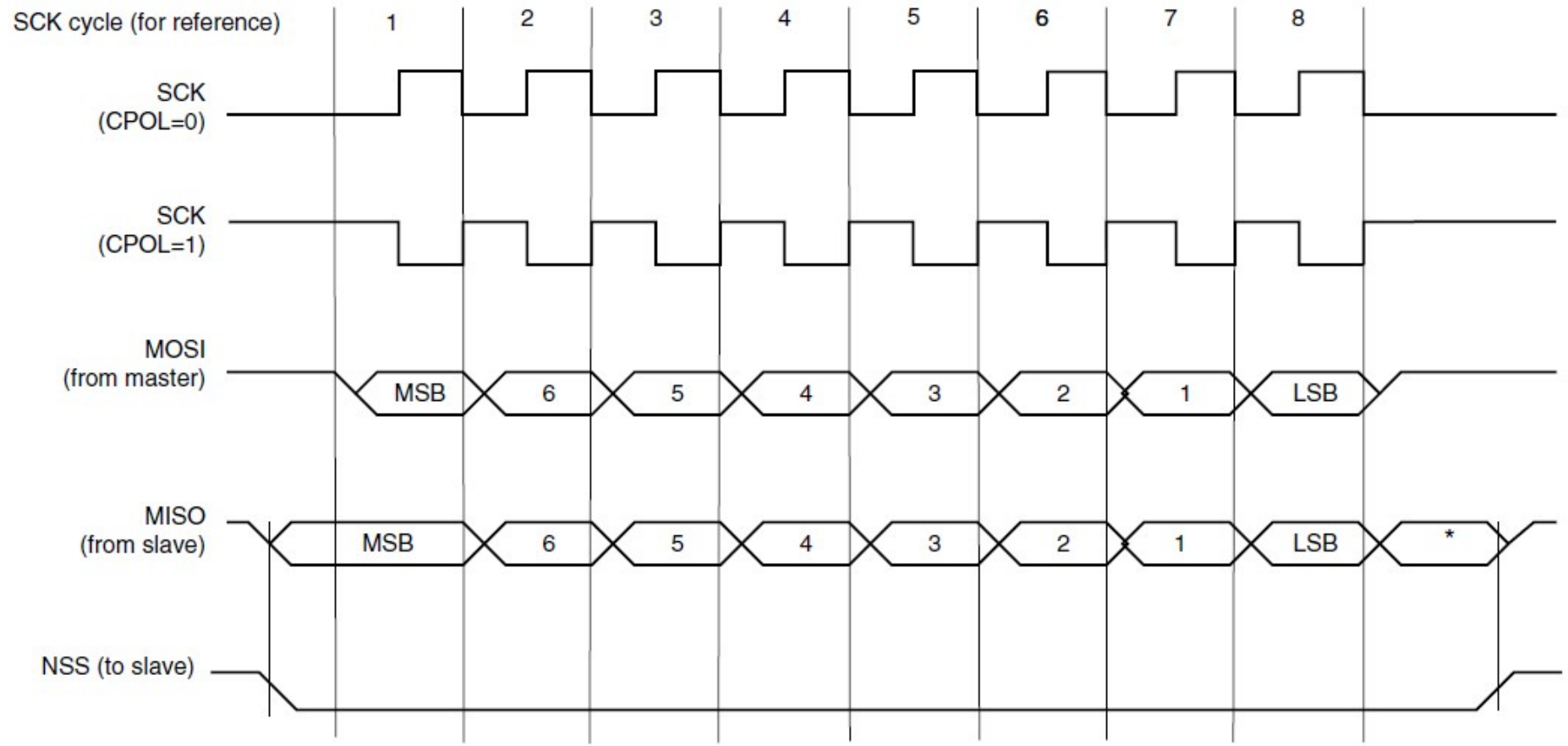
- Alternative naming conventions are also widely used:
  - SCK, CLK: Serial Clock (output from master)
  - SDI, DI, DIN, SI: Serial Data In; Data In, Serial In
  - SDO, DO, DOUT, SO: Serial Data Out; Data Out, Serial Out
  - nCS, CS, CSB, CSN, nSS, STE: Chip Select, Slave Transmit Enable (active low, output from master)



SPI: Data Transmission



SPI: Data Transmission (Example)



SPI: Data Transmission (1)

1. Master configures the clock
   - Frequency less than or equal to the maximum frequency the slave device supports.
   - Commonly in the range of 1–70 MHz.
2. Master pulls the chip select low for the desired chip.
3. If a waiting period is required (such as for analog-to-digital conversion) then the master must wait for at least that period of time before starting to issue clock cycles.

SPI: Data Transmission (2)

4. During each SPI clock cycle, a full duplex data transmission occurs:
   - Master sends a bit on the MOSI line; the slave reads it from that same line
   - Slave sends a bit on the MISO line; the master reads it from that same line

- Not all transmissions require all four of these operations to be meaningful but they do happen.

SPI: Data Transmission (3)

- Transmissions normally involve two shift registers of some given word size, such as eight bits
  - one in the master and one in the slave
  - they are connected in a ring.
- Data are usually shifted out with the most significant bit first, while shifting a new least significant bit into the same register.
- After that register has been shifted out, the master and slave have exchanged register values.

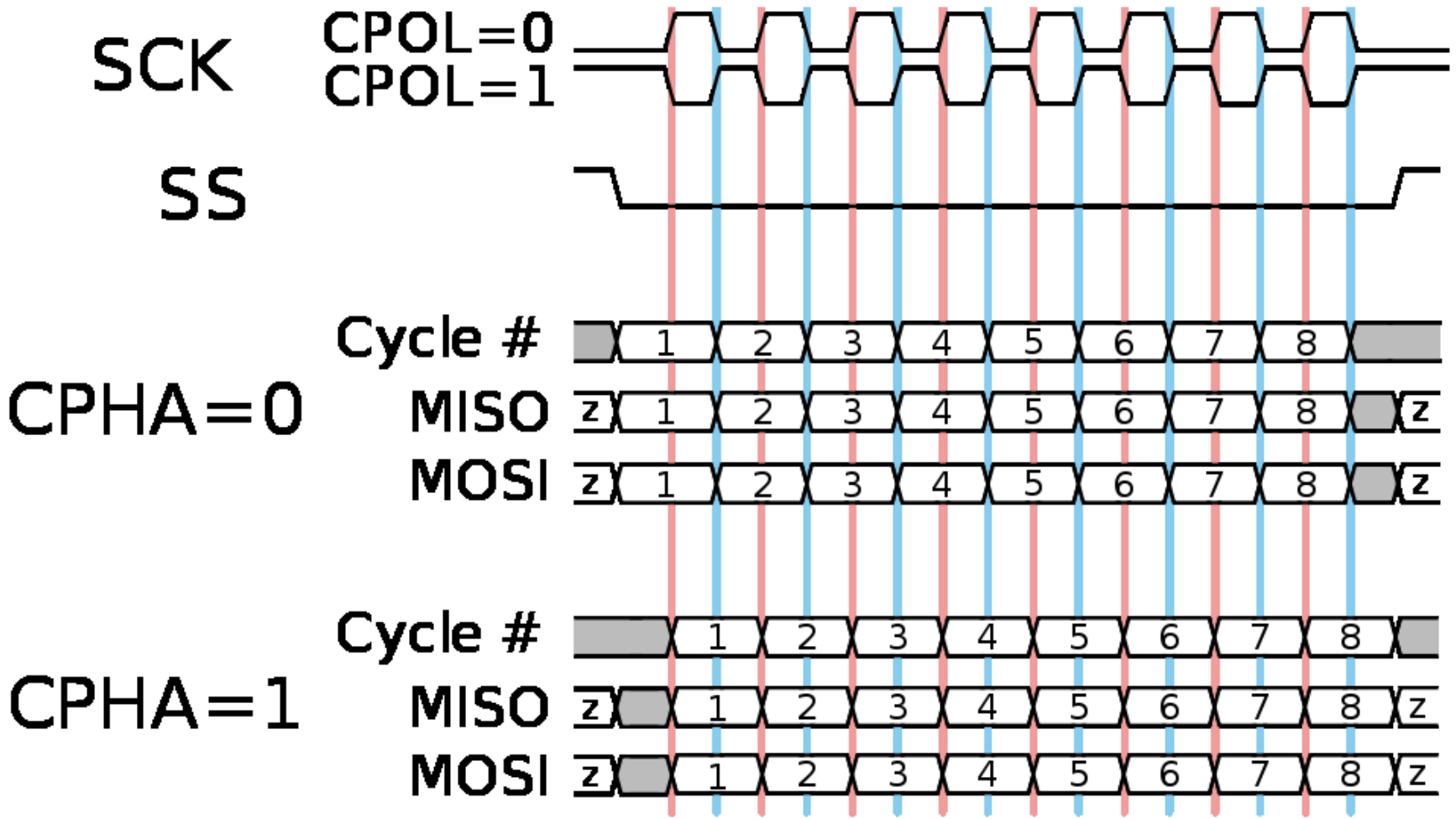
SPI: Data Transmission (4)

- Then each device takes that value and does something with it, such as writing it to memory.
- If there are more data to exchange, the shift registers are loaded with new data and the process repeats.
- Transmissions may involve any number of clock cycles.
- When there are no more data to be transmitted, the master stops toggling its clock. Normally, it then deselects the slave.

SPI: Data Transmission (5)

- Transmission word sizes
  - 8-bit words
  - 16-bit words for touch screen controllers or audio codecs,
  - 12-bit words for many DACs or ADCs.
- Non-selected slave on the bus:
  - must disregard the input clock and MOSI signals
  - must not drive MISO.
- The master must select only one slave at a time.

SPI: Clock Polarity and Phase (1)



SPI: Clock Polarity and Phase (2)

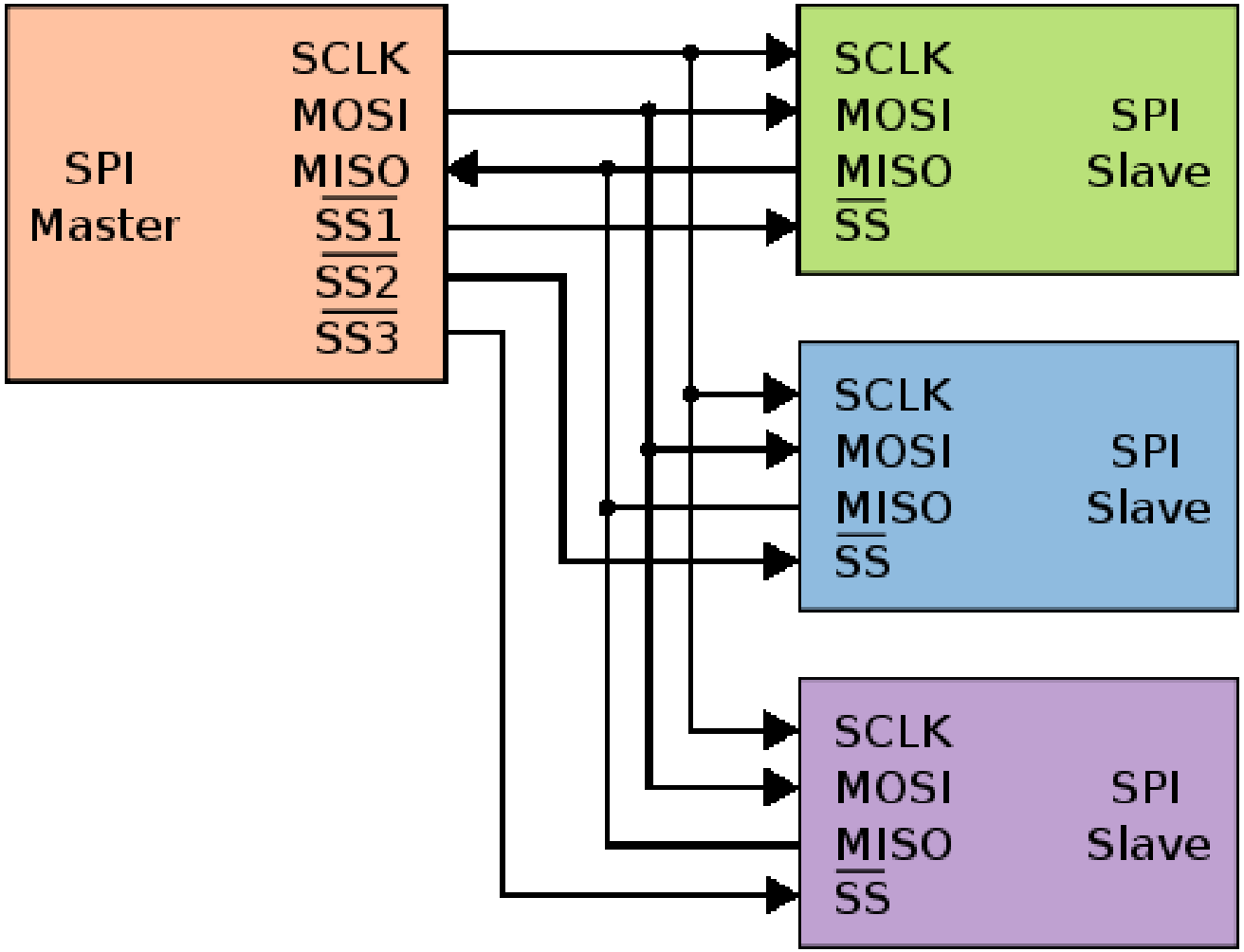
- At CPOL=0 the base value of the clock is zero
  - For CPHA=0, data are captured on the clock's rising edge (low→high transition) and data are propagated on a falling edge (high→low clock transition).
  - For CPHA=1, data are captured on the clock's falling edge and data are propagated on a rising edge.
- At CPOL=1 the base value of the clock is one (inversion of CPOL=0)
  - For CPHA=0, data are captured on clock's falling edge and data are propagated on a rising edge.
  - For CPHA=1, data are captured on clock's rising edge and data are propagated on a falling edge.

SPI: Clock Polarity and Phase (3)

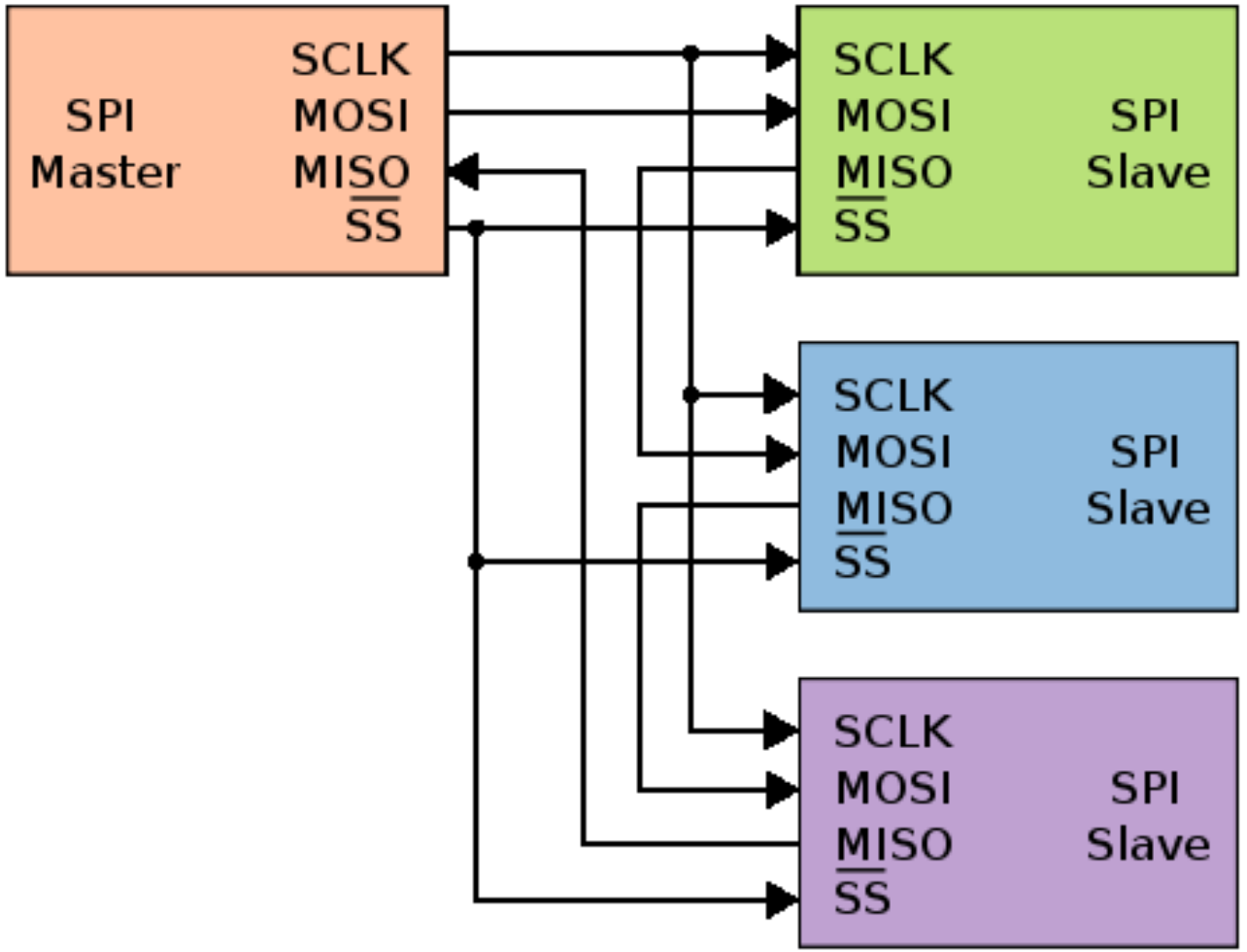
- Mode numbers
- The combinations of polarity and phases are often referred to as modes which are commonly numbered according to the following convention, with CPOL as the high order bit and CPHA as the low order bit:

Mode	CPOL	CPHA
0	0	0
1	0	1
2	1	0
3	1	1

Independent Slave SPI Configuration



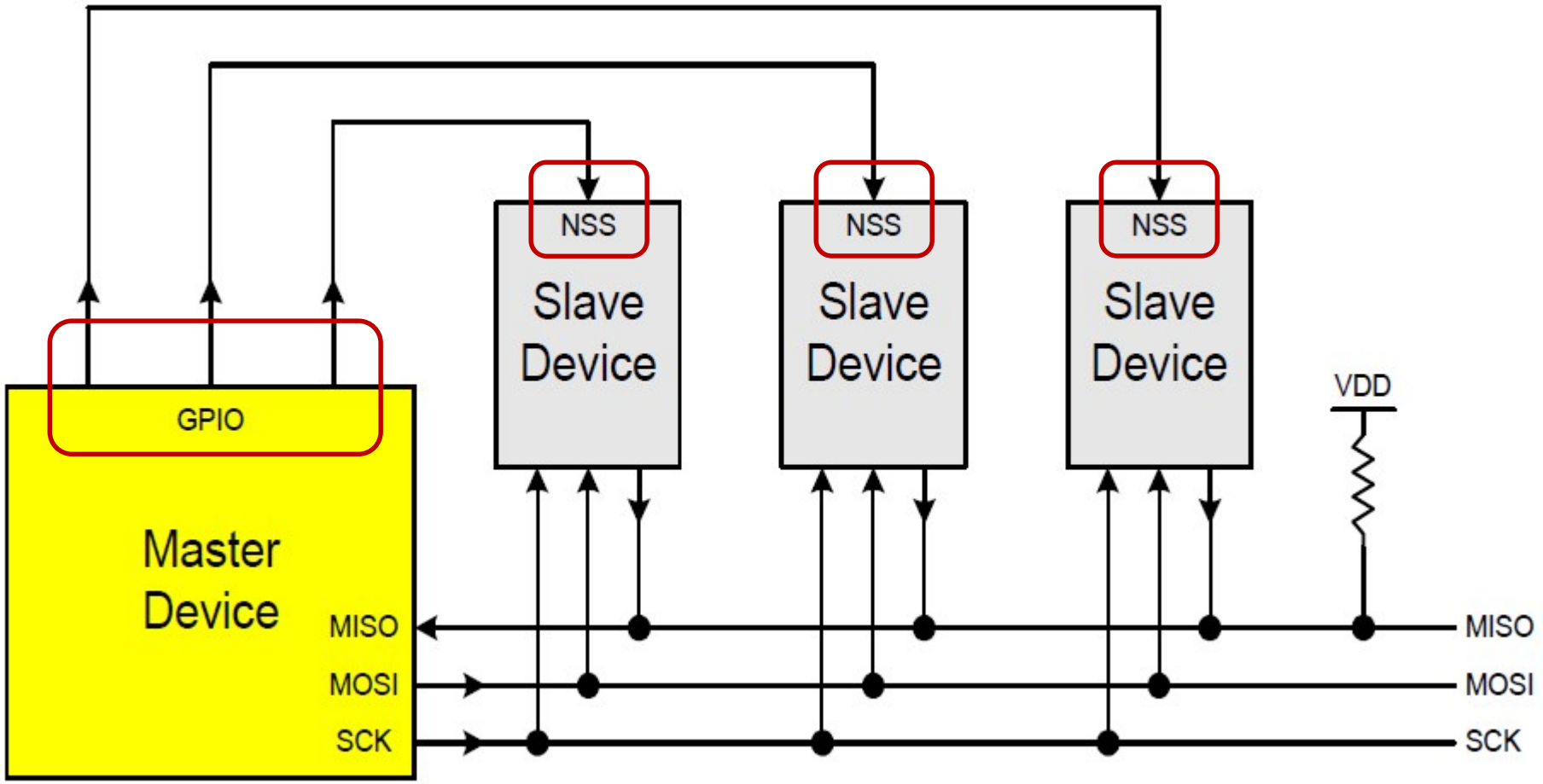
Daisy Chain SPI Configuration



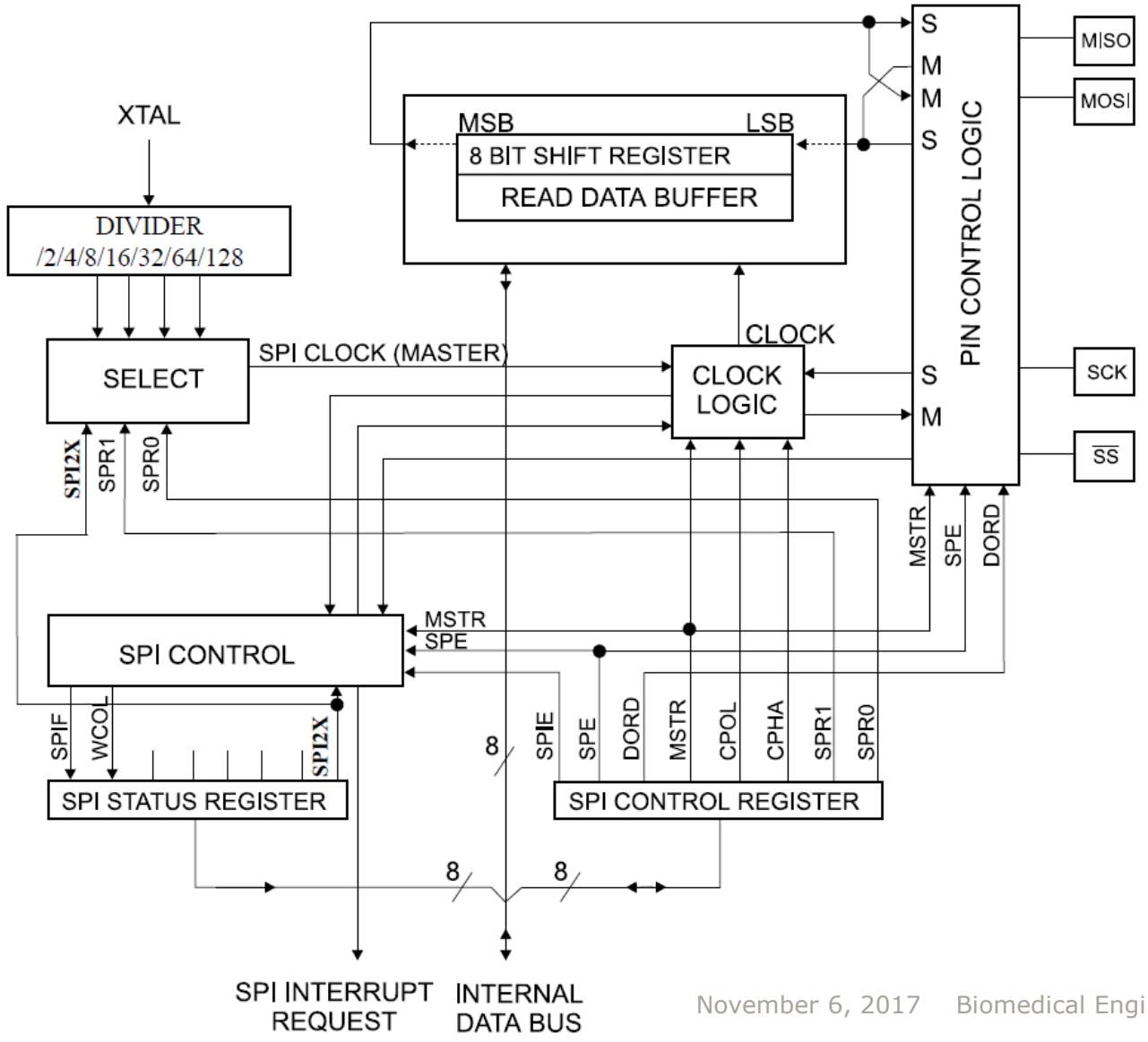
ATmega328PB SPI Features

- Full-duplex, Three-wire Synchronous Data Transfer
- Master or Slave Operation
- LSB First or MSB First Data Transfer
- Seven Programmable Bit Rates
- End of Transmission Interrupt Flag
- Write Collision Flag Protection
- Wake-up from Idle Mode
- Double Speed (CK/2) Master SPI Mode
- Two SPIs are available - `SPI0` and `SPI1` (ATmega328PB only)

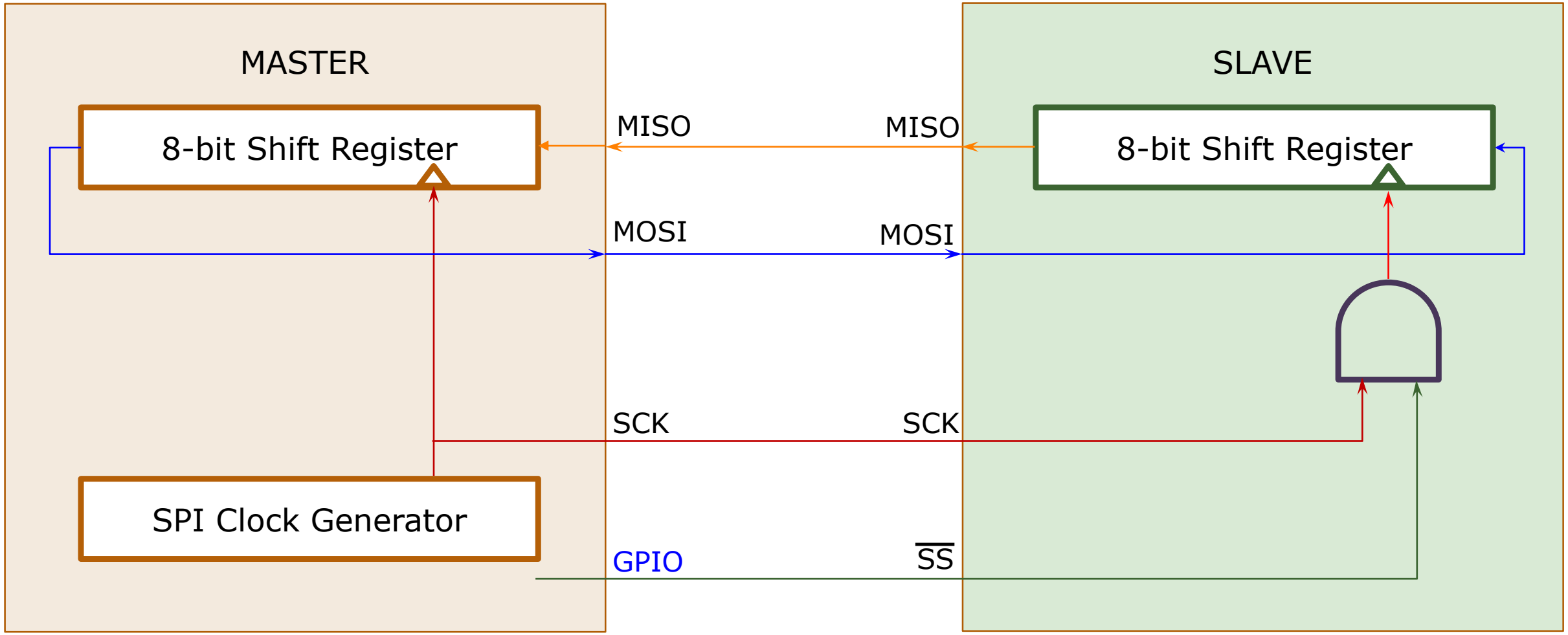
Typical SPI Interconnection



ATmega328PB SPI Block Diagram



ATmega328PB SPI Master-Slave Interconnection



ATmega328PB SPI Master-Slave Data Exchange

- The SPI Master initiates the communication cycle by pulling low the Slave Select, $\overline{SS}$, pin of the desired Slave.
- Master and Slave prepare the data to be sent in their respective shift Registers, and the Master generates the required clock pulses on the SCK line to interchange data.
- Data is always shifted from Master to Slave on the Master Out – Slave In, MOSI, line, and from Slave to Master on the Master In – Slave Out, MISO, line.
- After each data packet, the Master will synchronize the Slave by pulling high the Slave Select, $\overline{SS}$, line.

$\overline{SS}$ line at ATmega328PB SPI Master

- When configured as a Master, the SPI interface has no automatic control of the SS line.
  - This must be handled by user software before communication can start.
- When this is done, writing a byte to the SPI Data Register (SPDR) starts the SPI clock generator, and the hardware shifts the eight bits into the Slave.
- After shifting one byte, the SPI clock generator stops, setting the End of Transmission Flag (SPIF).
  - If the SPI Interrupt Enable bit (SPIE) in the SPCR Register is set, an interrupt is requested.
- The Master may continue to shift the next byte by writing it into SPDR, or signal the end of packet by pulling high the Slave Select ($\overline{SS}$) line.
- The last incoming byte will be kept in the SPI Data Register (SPDR) for later use.

$\overline{SS}$ line at ATmega328PB SPI Slave

- When configured as a Slave, the SPI interface will remain sleeping with MISO tri-stated as long as the $\overline{SS}$ pin is driven high.
- In this state, software may update the contents of the SPI Data Register (SPDR), but the data will not be shifted out by incoming clock pulses on the SCK pin until the $\overline{SS}$ pin is driven low.
- As one byte has been completely shifted, the End of Transmission Flag (SPIF) is set.
  - If the SPI Interrupt Enable bit (SPIE) in the SPCR Register is set, an interrupt is requested.
- The Slave may continue to place new data to be sent into SPDR before reading the incoming data.
- The last incoming byte will be kept in the SPI Data Register (SPDR) for later use.

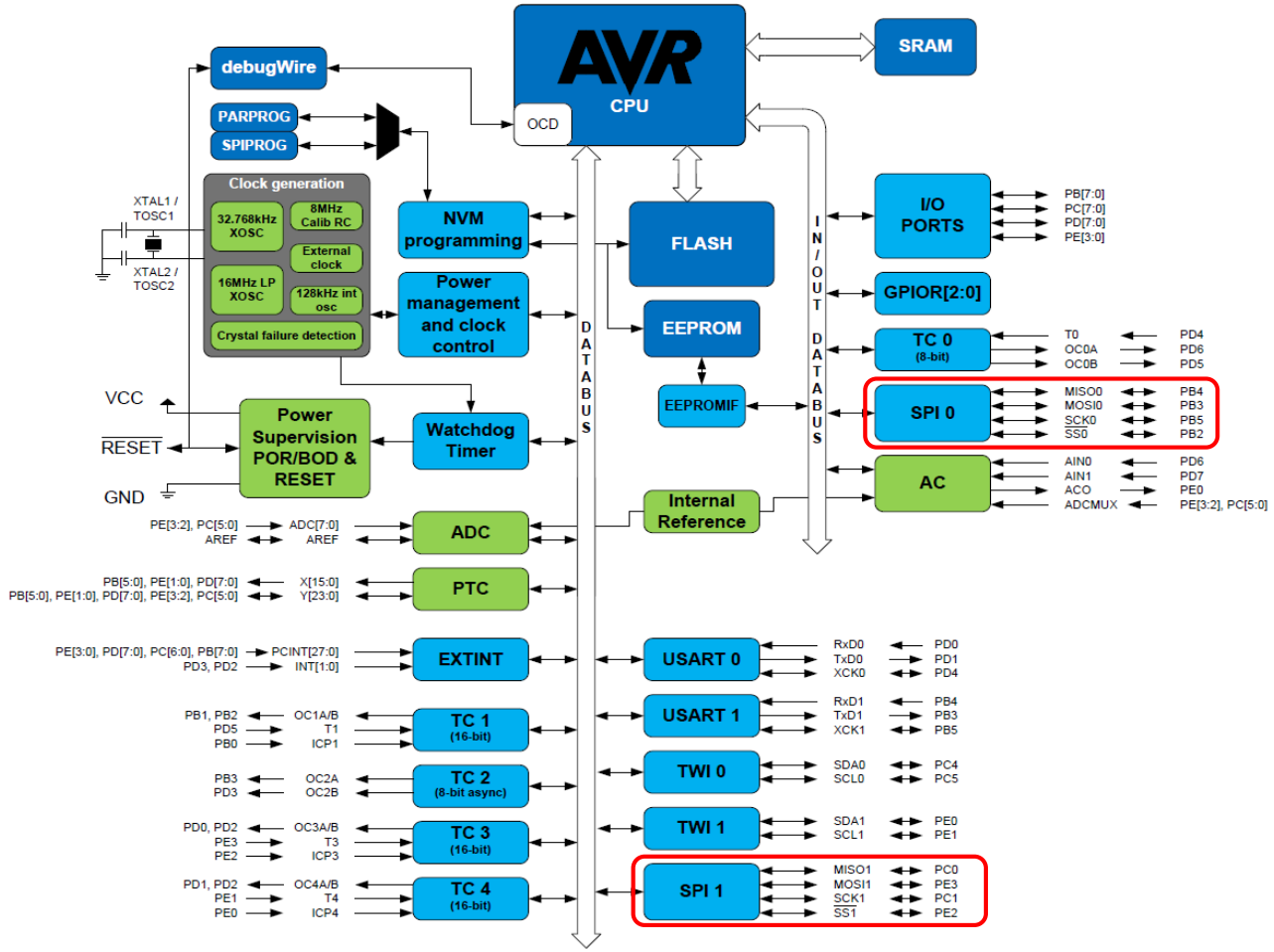
ATmega328PB SPI Transmit and Receive Buffering

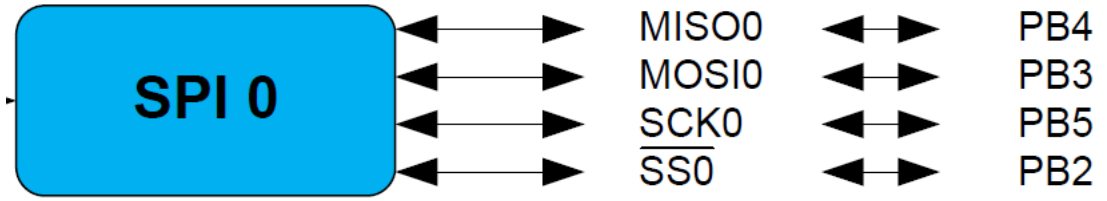
- The system is single buffered in the transmit direction and double buffered in the receive direction.
- This means that bytes to be transmitted cannot be written to the SPI Data Register (SPDR) before the entire shift cycle is completed.
- When receiving data, however, a received character must be read from the SPI Data Register(SPDR) before the next character has been completely shifted in. Otherwise, the first byte is lost.

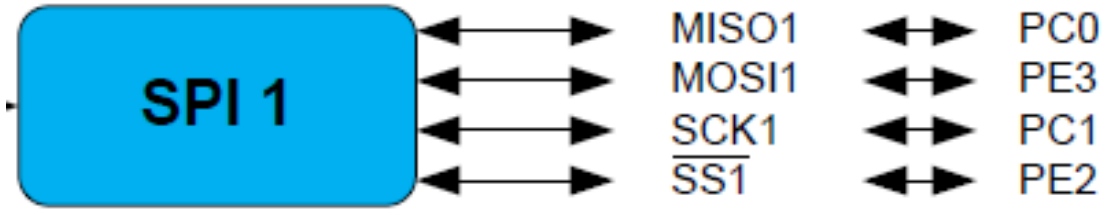
Sampling of Receiving Data in Slave Mode

- In SPI Slave mode, the control logic will sample the incoming signal of the SCK pin.
- To ensure correct sampling of the clock signal, the minimum low and high periods should be longer than two CPU clock cycles.

ATmega328PB Data Direction of MOSI, MISO, SCK, and SS Pins







ATmega328PB Data Direction of MOSI, MISO, SCK, and SS Pins (Master)

- When the SPI is enabled, the data direction of the MOSI, MISO, SCK, and SS pins is overridden according to the table below.

Pin	Direction, Master SPI
MOSI	User Defined
MISO	Input
SCK	User Defined
SS	User Defined

```
void SPI0_MasterInit(void)
{
    DDRB = (1 << DDB5)  // Set SCK to OUTPUT mode
         | (1 << DDB3)  // Set MOSI to OUTPUT mode
         | (1 << DDB2); // Set SS to OUTPUT mode

    SPCR0 = (1 << SPE0)    // Enable SPI0
          | (1 << MSTR0)   // Enable Master
          | (1 << SPR00);  // Set clock rate to fosc/16
}
```

ATmega328PB Example Codes for SPI Master

```
void SPI0_MasterInit(void)
{
    // Set SCK, MOSI and nSS output
    DDRB = (1 << DDB5) | (1 << DDB3) | (1 << DDB2);

    // Enable SPI, Master, set clock rate fosc/16
    SPCR0 = (1 << SPE0) | (1 << MSTR0) | (1 << SPR00);
}

void SPI0_MasterTransmit(char cData)
{
    // Start transmission
    SPDR0 = cData;

    // Wait for transmission complete
    while ((SPSR0 & (1 << SPIF0)) == 0)
        ;
}
```

ATmega328PB Data Direction of MOSI, MISO, SCK, and SS Pins (Slave)

- When the SPI is enabled, the data direction of the MOSI, MISO, SCK, and SS pins is overridden according to the table below.

Pin	Direction, Slave SPI
MOSI	Input
MISO	User Defined
SCK	Input
SS	Input

```
void SPI0_SlaveInit(void)
{
    // Set MISO output
    DDRB = (1 << DDB4);

    // Enable SPI0
    SPCR0 = (1 << SPE0);
}
```

ATmega328PB Example Codes for SPI Slave

```
void SPI0_SlaveInit(void)
{
    // Set MISO output
    DDRB = (1 << DDB4);

    // Enable SPI0
    SPCR0 = (1 << SPE0);
}

char SPI0_SlaveReceive(void)
{
    // Wait for reception complete
    while ((SPSR0 & (1 << SPIF0)) == 0);

    // Return received data
    return SPDR0;
}
```

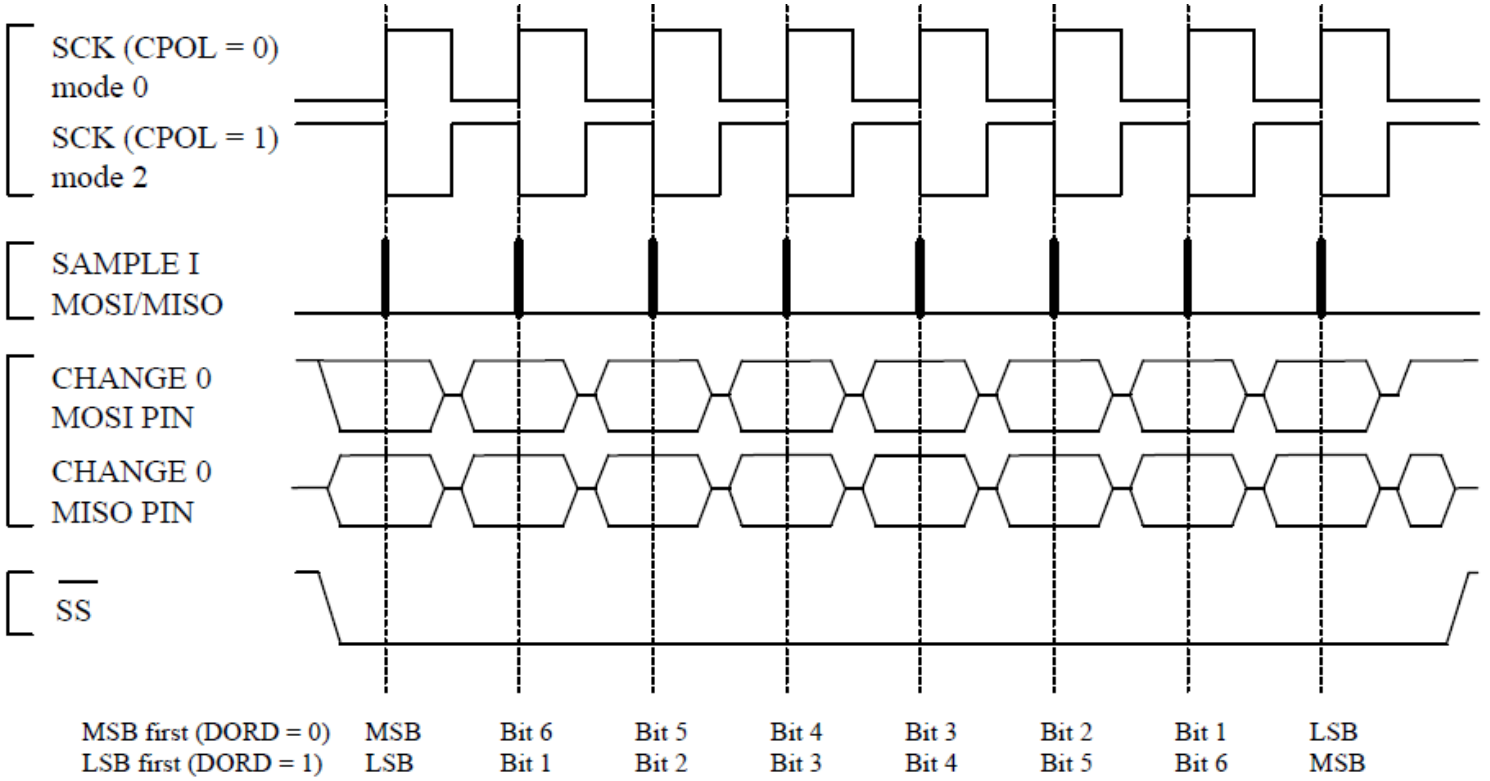
ATmega328PB Data Modes (1)

- There are four combinations of SCK phase and polarity with respect to serial data, which are determined by control bits CPHA and CPOL.
- Data bits are shifted out and latched in on opposite edges of the SCK signal, ensuring sufficient time for data signals to stabilize.
- The following table, summarizes SPCR.CPOL and SPCR.CPHA settings.

SPI Modes	Conditions	Leading Edge	Trailing Edge
0	CPOL=0, CPHA=0	Sample on Rising Edge	Setup on Falling Edge
1	CPOL=0, CPHA=1	Setup on Rising Edge	Sample on Falling Edge
2	CPOL=1, CPHA=0	Sample on Falling Edge	Setup on Rising Edge
3	CPOL=1, CPHA=1	Setup on Falling Edge	Sample on Rising Edge

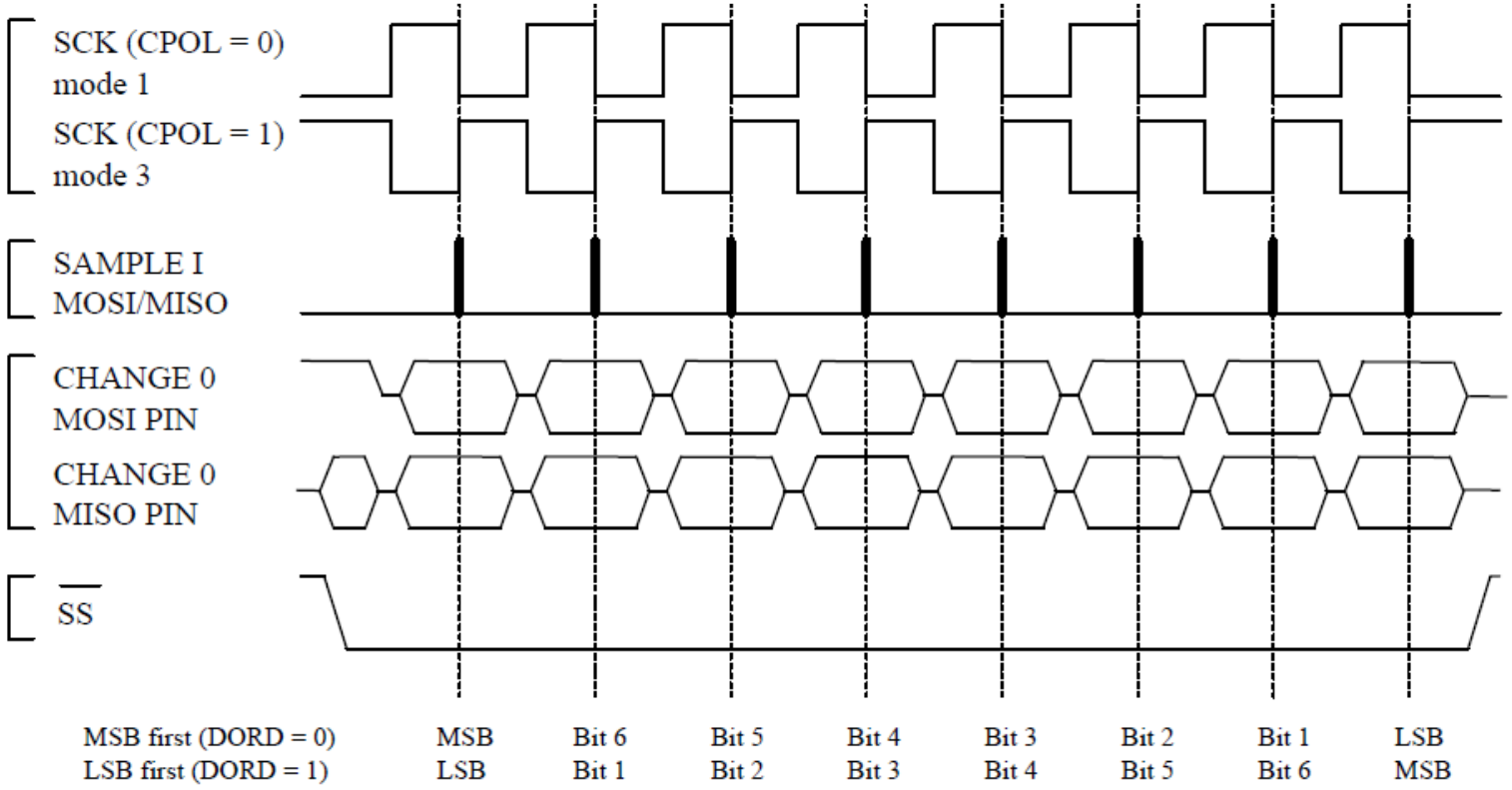
ATmega328PB Data Modes (2)

SPI Modes	Conditions	Leading Edge	Trailing Edge
0	CPOL=0, CPHA=0	Sample on Rising Edge	Setup on Falling Edge
2	CPOL=1, CPHA=0	Sample on Falling Edge	Setup on Rising Edge

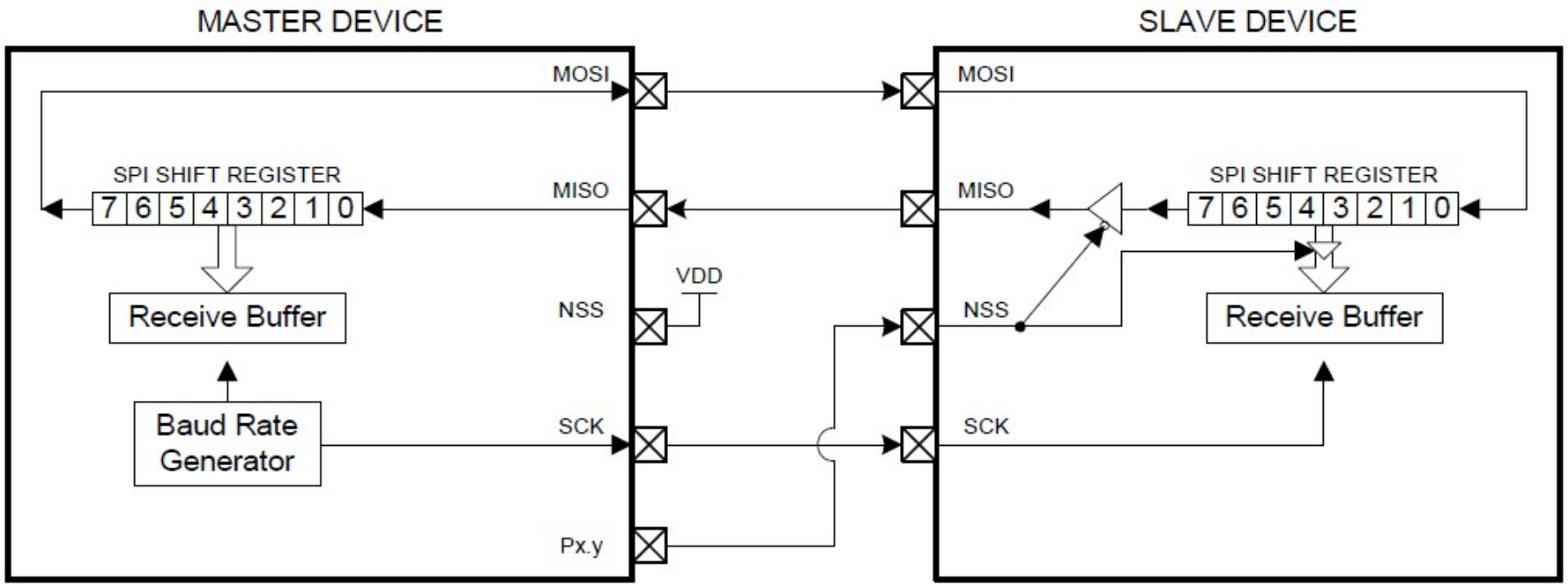


ATmega328PB Data Modes (3)

SPI Modes	Conditions	Leading Edge	Trailing Edge
1	CPOL=0, CPHA=1	Setup on Rising Edge	Sample on Falling Edge
3	CPOL=1, CPHA=1	Setup on Falling Edge	Sample on Rising Edge

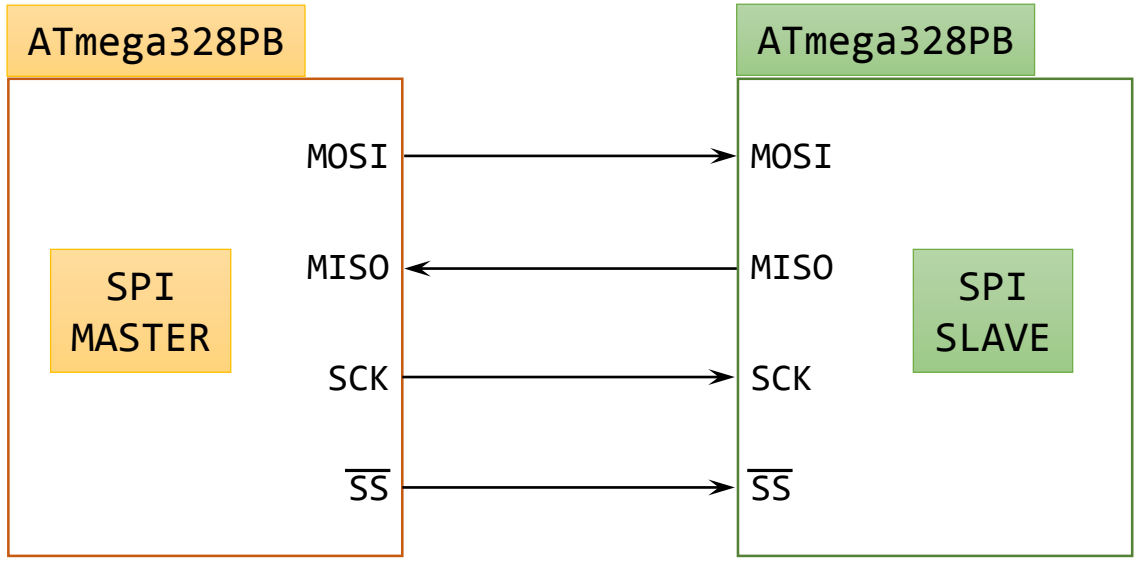


ATmega328PB SPI0 Operation



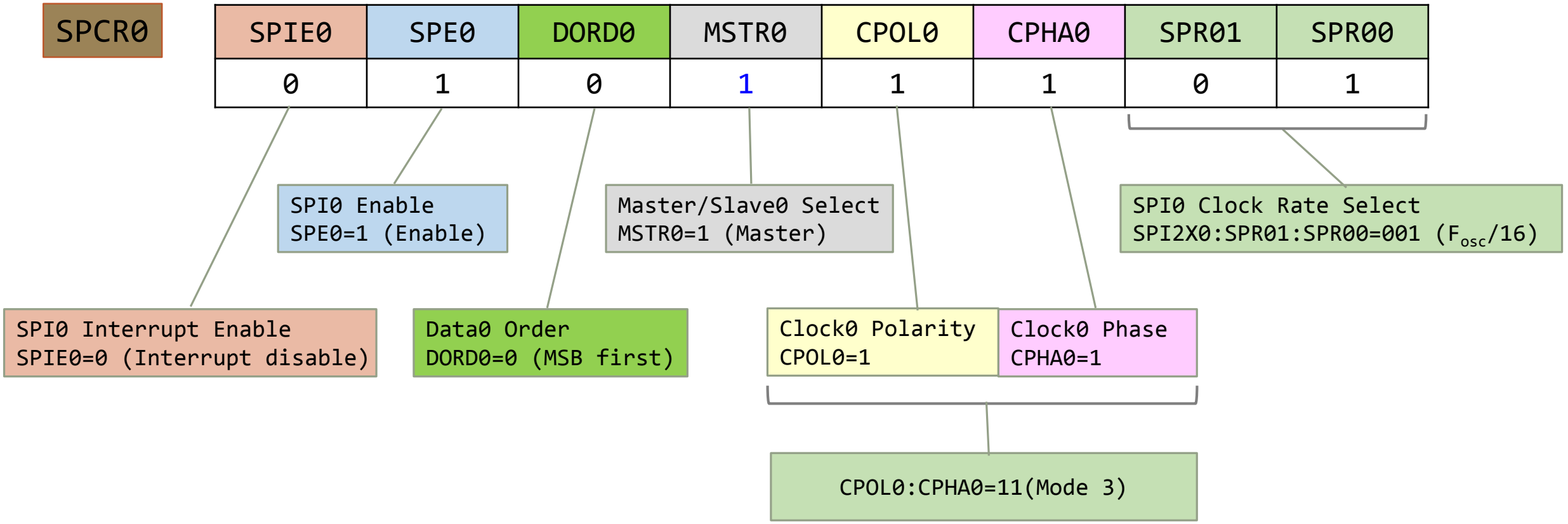
ATmega328PB SPI0 Example I (1)

- Specifications:
  - SCK: 1 MHz
  - MSB first
  - Mode 3
- MASTER sends SLAVE a byte data 0x55 and stores the received data to the variable `rx_data`.
- SLAVE sends MASTER a byte data 0xAA and stores the received data to the variable `rx_data`.
- Polling method



ATmega328PB SPI0 Example I (2)

1 MHz, MSB first, Mode 3



ATmega328PB SPI0 Example I (3)

1 MHz, MSB first, Mode 3

SPSR0

SPIF0	WCOL0						SPI2X0
0/1	1						0

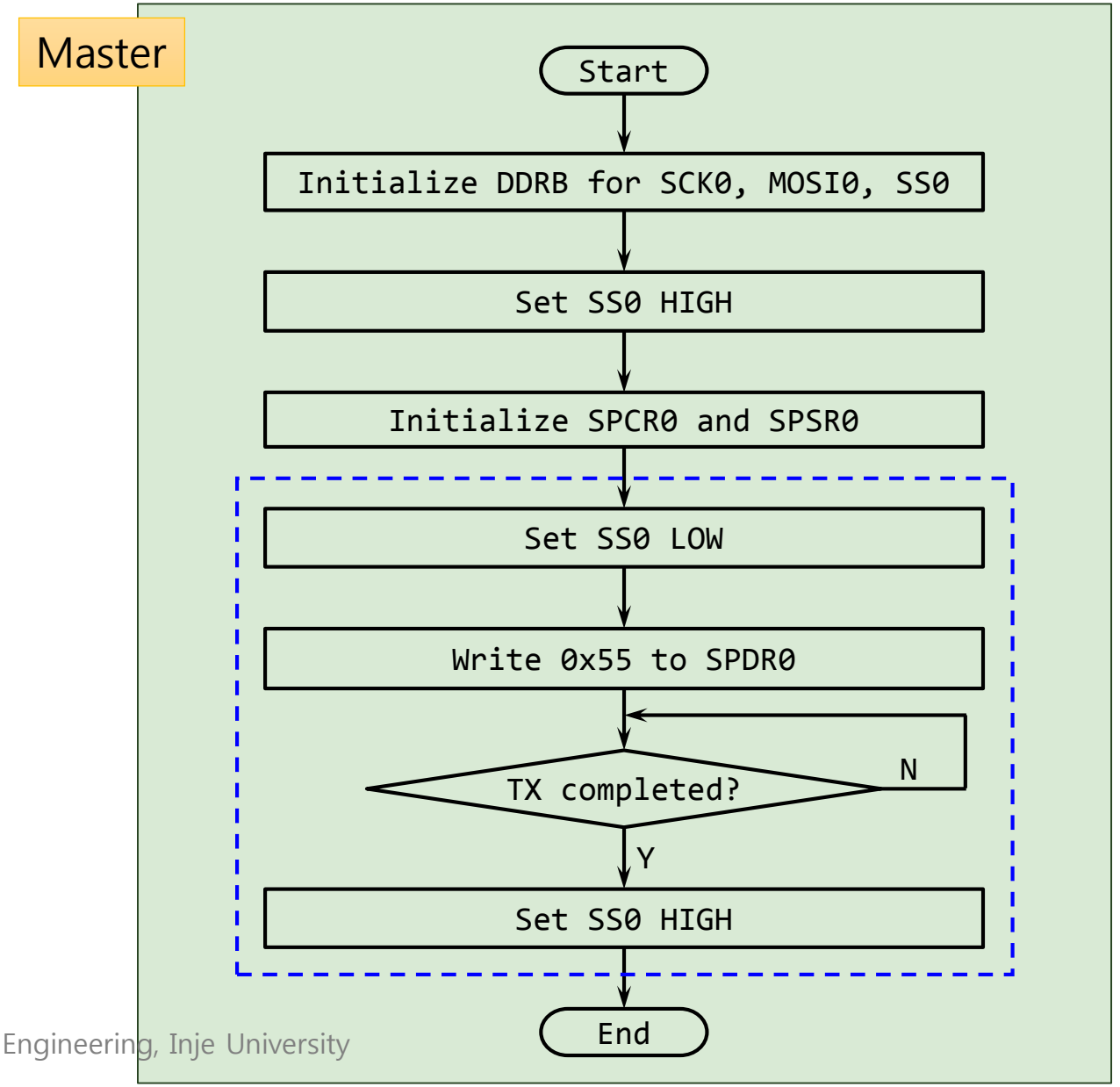
SPI0 Write Collision Flag
WCOL0=1 (SPDR is written during a data transfer)

SPI0 Clock Rate Select
SPI2X0:SPR01:SPR00=001 ($_{Fosc}$/16)

SPI0 Interrupt Flag
SPIE0=0 (Transfer is incomplete)
SPIE0=1 (Transfer is complete)

ATmega328PB SPI0 Example I(Master)(4)

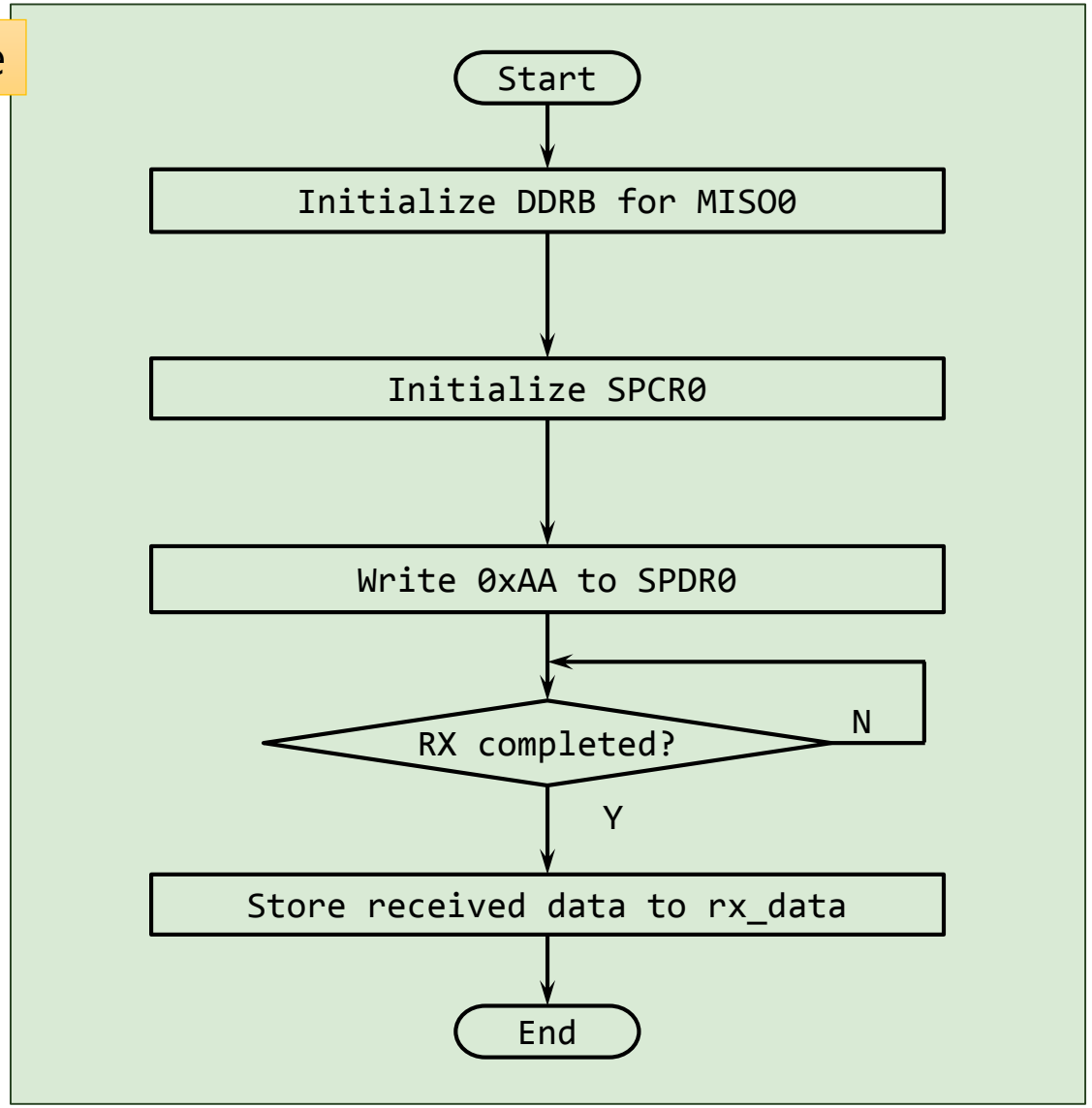
- Specifications:
  - SCK: 1 MHz
  - MSB first
  - Mode 3
- MASTER sends SLAVE a byte data 0x55 and stores the received data to the variable `rx_data`.
- SLAVE sends MASTER a byte data 0xAA and stores the received data to the variable `rx_data`.
- Polling method



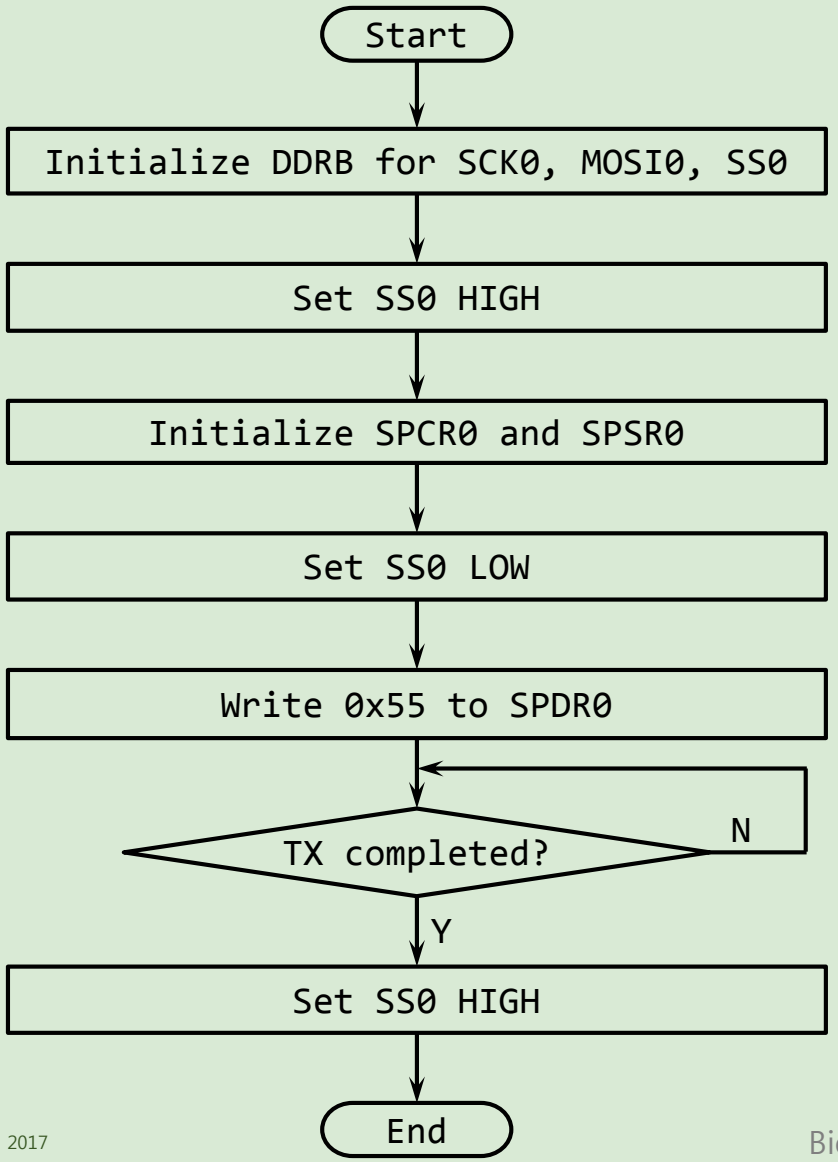
ATmega328PB SPI0 Example I(Slave)(5)

- Specifications:
  - SCK: 1 MHz
  - MSB first
  - Mode 3
- MASTER sends SLAVE a byte data 0x55 and stores the received data to the variable `rx_data`.
- SLAVE sends MASTER a byte data 0xAA and stores the received data to the variable `rx_data`.
- Polling method

Slave

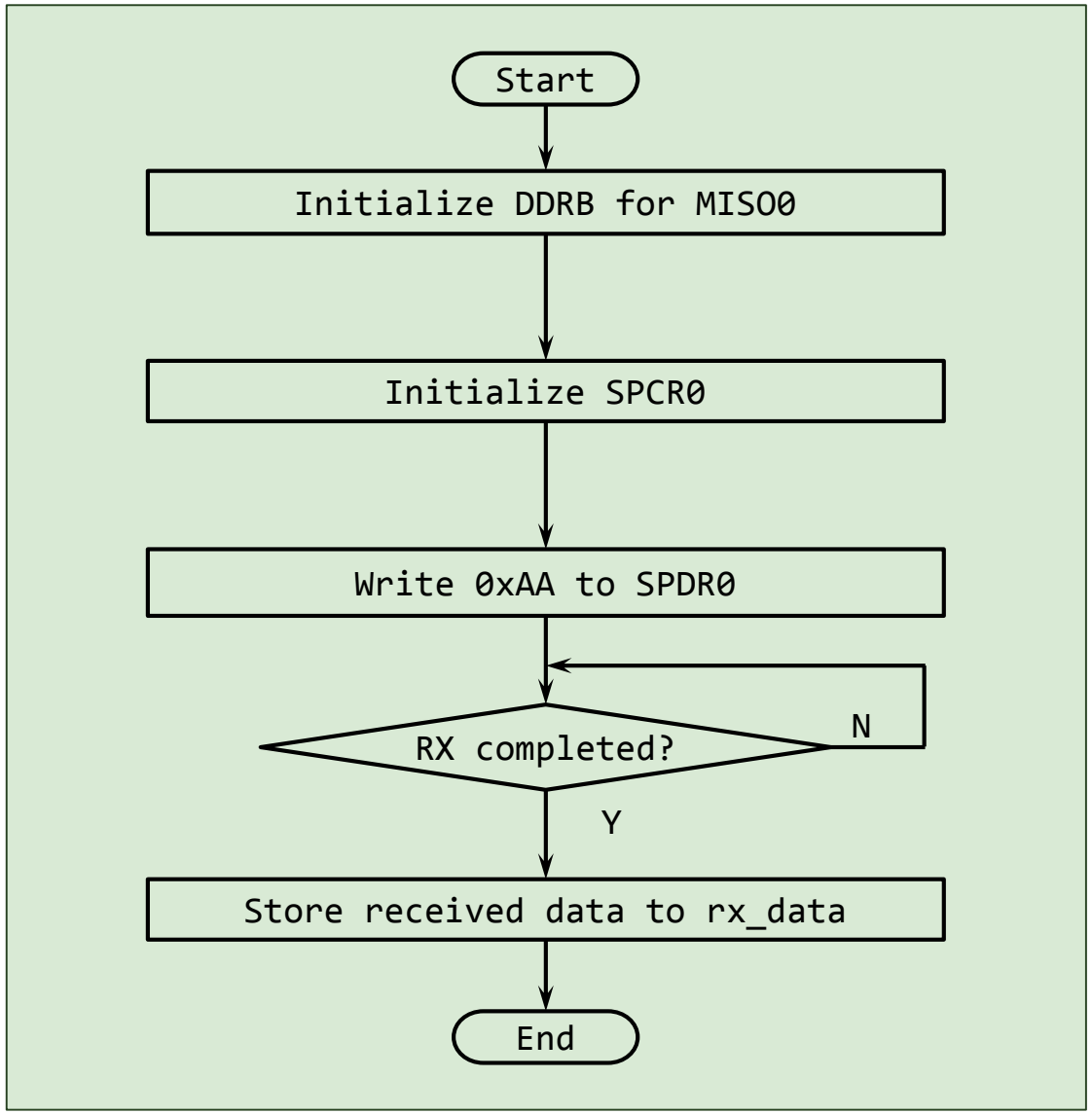


ATmega328PB SPI0 Example I(Master)(6)




```c
#include <avr/io.h>

int main(void)
{
    char rx_data;

    // Init pins for SPI0
    DDRB = (1 << DDRB5)     // Set PB5 as OUTPUT for SCK0
         | (1 << DDRB3)     // Set PB3 as OUTPUT for MOSI0
         | (1 << DDRB2);    // Set PB2 as OUTPUT for SS0

    // Set SS0 HIGH
    PORTB |= (1 << PORTB2);

    // Init SPI0 as a MASTER
    SPCR0 = (1 << SPE)        // Enable SPI0
          | (1 << MSTR)       // Select MASTER
          | (1 << CPOL)       // CPOL = 1 (Mode 3)
          | (1 << CPHA)       // CPHA = 1 (Mode 3)
          | (1 << SPR0);      // SPI0 clock = 16MHz/16 = 1MHz.

    PORTB &= ~(1 << PORTB2);    // Set SS0 LOW

    SPDR0 = 0x55;               // Send a byte data
    while (!(SPSR0 & (1 << SPIF)));    // Wait for end of TX

    PORTB |= (1 << PORTB2);             // Set SS0 HIGH

    rx_data = SPDR0;        // Store the received data

    while (1)
    { }
}
```

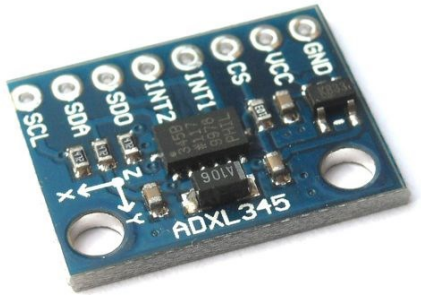
ATmega328PB SPI0 Example I(Slave)(7)




```c
#include <avr/io.h>

int main(void)
{
    char rx_data;

    // Init pins for SPI0
    DDRB = (1 << DDRB4);    // Set PB4 as OUTPUT for MISO0

    // Init SPI0 as a MASTER
    SPCR0 = (1 << SPE)      // Enable SPI0
          | (1 << CPOL)     // CPOL = 1 (Mode 3)
          | (1 << CPHA);    // CPHA = 1 (Mode 3)

    SPDR0 = 0xAA;           // Load a byte data
    while (!(SPSR0 & (1 << SPIF)));   // Wait for end of reception

    rx_data = SPDR0;        // Store the received data

    while (1)
    { }
}
```

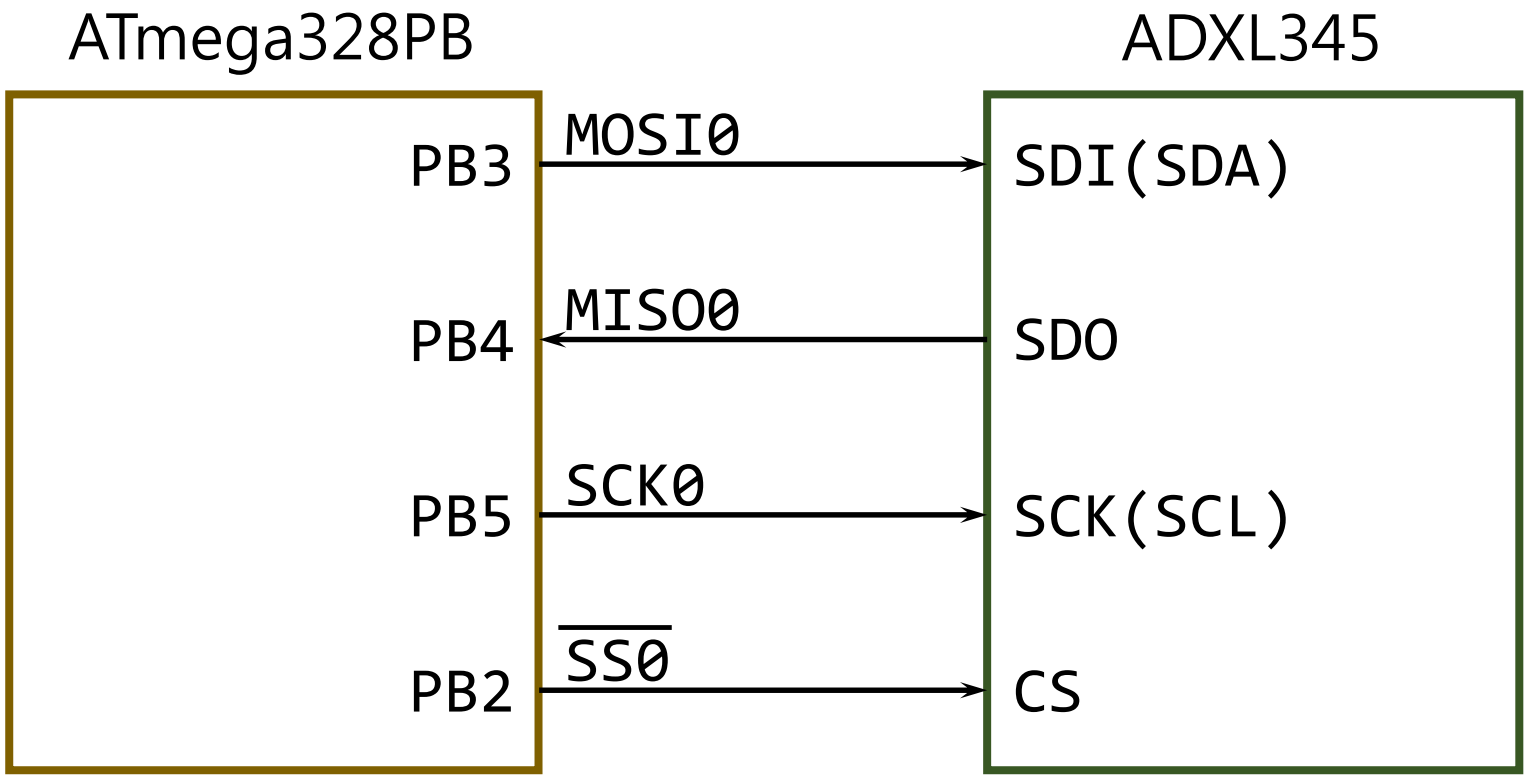
ATmega328PB SPI0 Example II (1)

- ADXL345: 3-axis accelerometer
- Interface ADXL345 using SPI0
- Initialize ADXL345
- Read data for X-, Y-, and Z-axis from ADXL345 and transmit the data through UART0.
- Polling method



ATmega328PB SPI0 Example II (2)

Interface ADXL345 using SPI0



ATmega328PB SPI0 Example II (3)

Read ADXL345 ID

Register Address	Name	Type	Reset Value	Description
0x00	DEVID	R	11100101	Device ID

ATmega328PB SPI0 Example II (4)

Initialize ADXL345

Register Address	Name	Type	Set Value	Description
0x2C	BW_RATE	R/$\overline{W}$	00001010	BW=50Hz (Output Data Rate = 100Hz)
0x31	DATA_FORMAT	R/$\overline{W}$	00001000	Full Resolution +/-2g, 4mg/LSB
0x2D	POWER_CTL	R/$\overline{W}$	00001000	Enter Measurement Mode

ATmega328PB SPI0 Example II (5)

Read ADXL345 Status (Address: 0x30, Read Only)

Bit 7 of INT_SOURCE Register (Address 0x30)

1: Data is available

0: Data is not available

ATmega328PB SPI0 Example II (6)

Read Data (X-, Y-, Z-Axis) from ADXL345

- The output data is two's complement format.
- It is recommended that a multiple-byte read of all registers be performed to prevent a change in data between reads of sequential registers.

Register Address	Name	Type	Reset Value	Description
0x32	DATAX0	R	00000000	Low Byte of X-Axis Data
0x33	DATAX1	R	00000000	High Byte of X-Axis Data
0x34	DATAY0	R	00000000	Low Byte of Y-Axis Data
0x35	DATAY1	R	00000000	High Byte of Y-Axis Data
0x36	DATAZ0	R	00000000	Low Byte of Z-Axis Data
0x37	DATAZ1	R	00000000	High Byte of Z-Axis Data

ATmega328PB SPI0 Example II (7)

```c
#include <avr/io.h>
#include <stdio.h>

int main(void)
{
    // Init pins for SPI0 (DDRB)

    // Set ADXL CS signal to High (PB2)

    // Init SPI0 as a MASTER (SPCR)

    // Read ADXL345 ID and display it

    // Set ADXL345 BW_RATE

    // Set ADXL345 DATA_FORMAT
    // Full Resolution, +/-2g (4mg/LSB)

    // Enter MEASUREMENT mode

    while (1)
    {
        // Wait until new data is available

        // Read 3-axis data with multiple-byte read format

        // Display 3-axis data

    }
}
```

```c
// Refer to Fig. 37 of ADXL345 Datasheet
void write_adxl345(uint8_t reg_addr, uint8_t data)
{

}

uint8_t read_adxl345_reg(uint8_t reg_addr)
{

}

void read_adxl345_reg_multi(uint8_t num, uint8_t
start_addr, uint8_t *buff)
{

}
```

Complete the code by yourself.

SPI

END