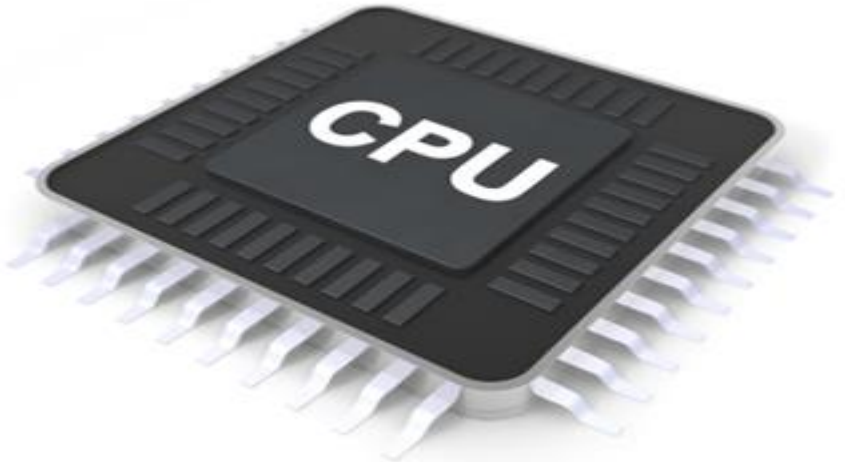


Serial Peripheral Interface

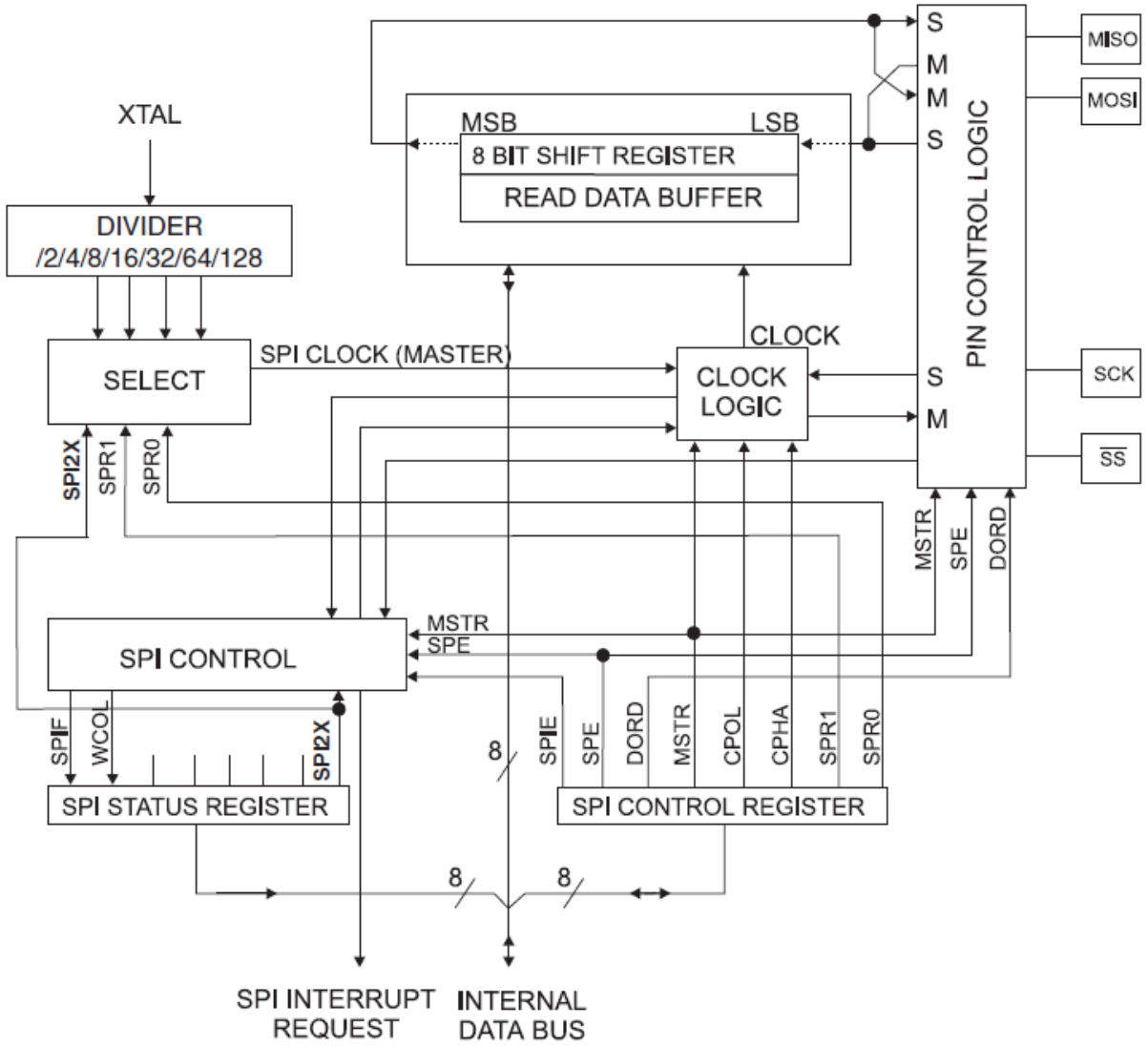
ENEE 3587

Microp Interfacing

SPI

- Synchronous,
  - Clock is transmitted (pin)
  - Good for short distance interfacing
- serial protocol,
  - 4-wire, 3-wire
  - A single device (“master”) is allowed to sync other devices (“slaves”)
- for peripheral interfacing,
  - Peripheral devices: such as sift registers, sensors, RTC, LCDs, DTA, DACs, etc.
  - Single or multiple peripherals
  - Can be used to interface MCUs
- developed by Freescale
  - Aka Motorola
  - Widely adopted

SPI Module


XTAL
DIVIDER
/2/4/8/16/32/64/128
SELECT
SPI CLOCK (MASTER)
MSB
LSB
8 BIT SHIFT REGISTER
READ DATA BUFFER
CLOCK
CLOCK LOGIC
S
M
M
S
PIN CONTROL LOGIC
S
M
MISO
MOSI
SCK
SS
SPI2X
SPR1
SPR0
MSTR
SPE
DORD
SPI CONTROL
MSTR
SPE
SPIF
WCOL
SPI2X
8
SPIE
SPE
DORD
MSTR
CPOL
CPHA
SPR1
SPR0
SPI STATUS REGISTER
SPI CONTROL REGISTER
8
8
SPI INTERRUPT REQUEST
INTERNAL DATA BUS

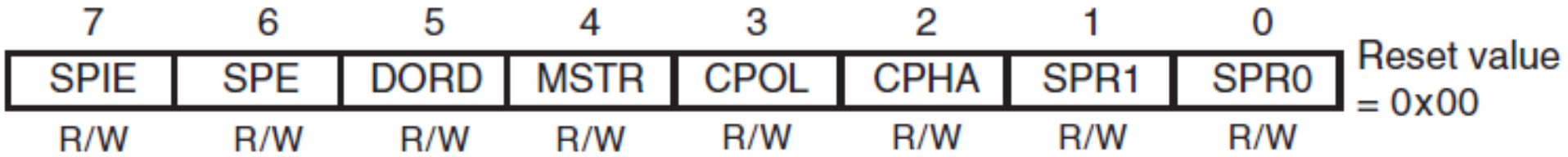
SPI Pins & Registers

- ATmega has 1 SPI
  - USART can be used in master SPI mode
    - USART pins are SCLK, TXD (master output), RXD (master input)
    - No SS
- SPI requires a max of 4 pins:
  - SCLK: serial clock: output from master, input slave → PB3
  - MISO: master input, slave output → PB2
  - MOSI: master input, slave output → PB1
  - $\overline{SS}$: slave select: output from master, input low for slave → PB0
- Registers:
  - SPCR: SPI control register (setup)
  - SPSR: SPI status register (flags)
  - SPDR: SPI data register (data)

Master/Slave

- Old terms baked into datasheets
  - Alternatives: controller- peripheral; chief-worker primary-secondary, etc.
  - https://en.wikipedia.org/wiki/Master/slave_(technology)
- Master:
  - Generates clock for other devices
  - Any single device can act as master
  - Uses **MI**SO as input; **MO**SI as output
  - SS pin output = 0
- Slave:
  - Clock is input
  - Multiple devices can be “slaves”
  - Uses MI**SO** as output; MO**SI** as input
  - SS pin input = 1

Mega SPI control register (SPCR)



SPIE: SPI interrupt enable

0 = Disable SPI interrupt

1 = Enable SPI interrupt.

SPE: SPI enable

0 = SPI disabled

1 = SPI enabled

DORD: Data order

0 = Most significant bit is transferred first.

1 = Least significant bit is transferred first.

MSTR: Master/slave select

0 = SPI module operates in slave mode.

1 = SPI module operates in master mode.

CPOL: Clock polarity

0 = SCK is idle low and hence its leading edge is rising and trailing edge is falling.

1 = SCK is idle high and hence its leading edge is falling and trailing edge is rising.

CPHA: Clock phase

0 = Samples data on the leading edge and allows data to setup on the trailing edge

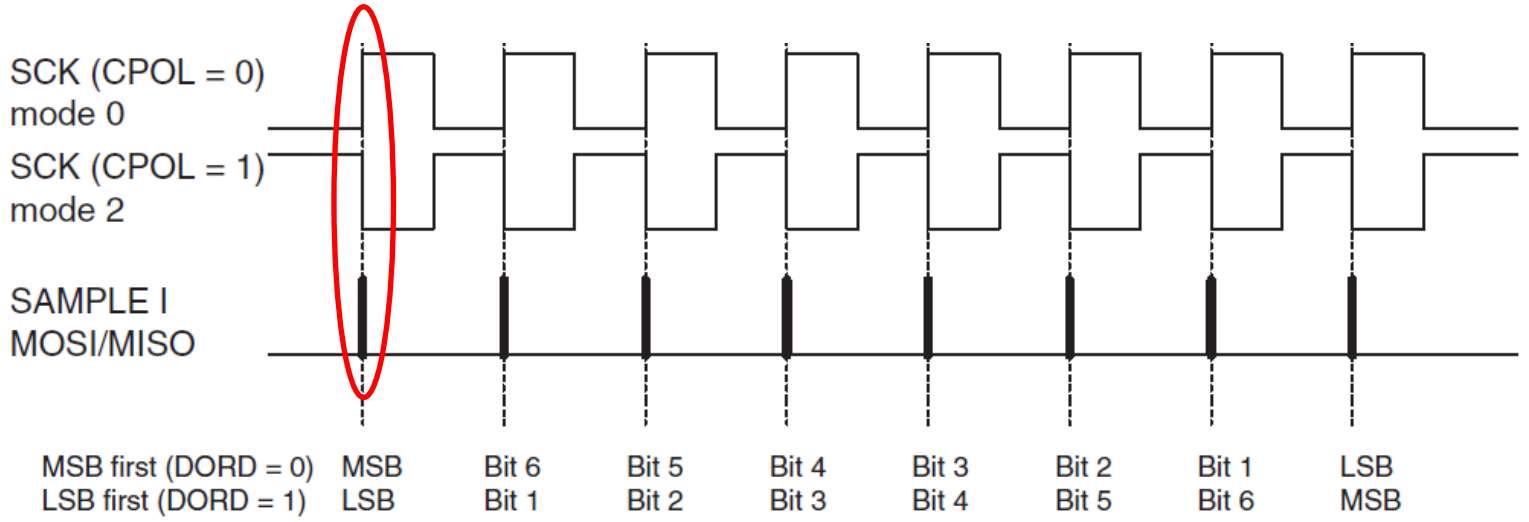
1 = Samples data on the trailing edge and allows data to setup on the leading edge

SPR1:SPR0: SPI clock rate select 1 and 0

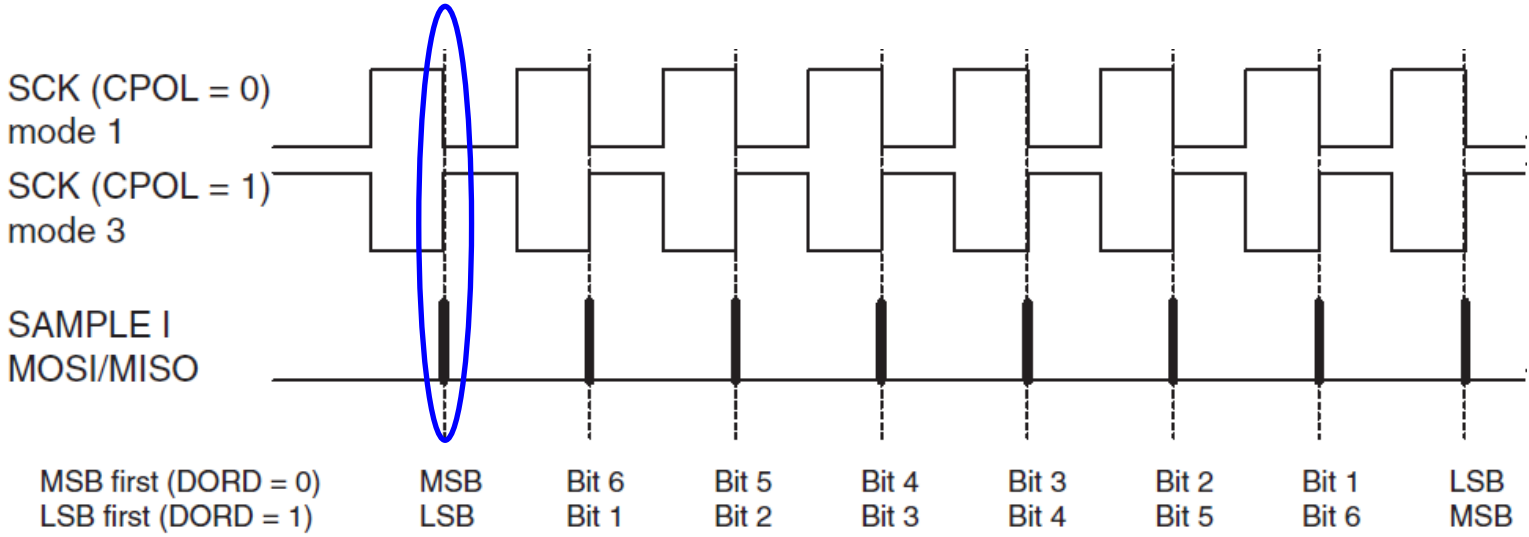
control the SCK rate of the device configured as a master.

CPOL, CPHA, DORD

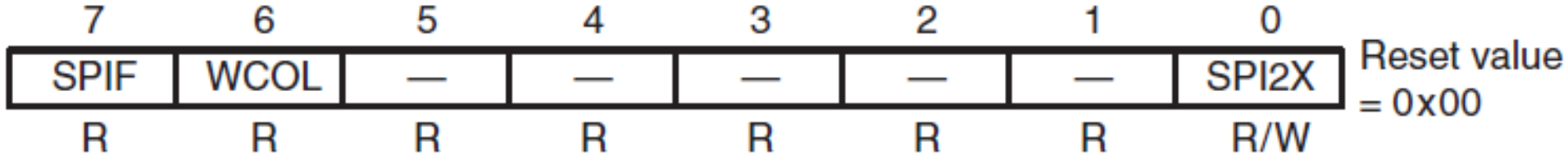
- CPHA=0: sample on lead edge
- CPOL=0: idle LO
- CPOL=1: idel HI
- DORD=0: shift MSB first
- DORD=1: shift LSB first



- CPHA=1: sample on trailing edge
- CPOL=0: idle LO
- CPOL=1: idel HI
- DORD=0: shift MSB first
- DORD=1: shift LSB first



SPI Status Register (SPSR)



- **SPIF**: SPI interrupt flag
  - This flag is set when a serial transfer is complete. If SS is an input and is driven low when the SPI is in master mode, this flag will also set this flag. SPIF is cleared when the corresponding interrupt service routine is entered. This flag can also be cleared by first reading the SPSR register and then accessing the SPDR register.
- **WCOL**: Write collision flag
  - This flag is set if the SPDR is written during a data transfer. This flag can be cleared by first reading SPSR register and then accessing the SPDR register.
- **SPI2X**: Double SPI speed bit
  - 0 = SPI data rate is normal (set by the SPR1: SPR0 bits).
  - 1 = SPI data rate is doubled As shown in Table 14.1. When SPI is in slave mode, it is guaranteed to work at fOSC/4 or lower only.

SPI Configuration

- MCU is typically the “Master”
  - Clock generation is more easily done by MCU that by peripheral
  - Input: MISO (PB2)
  - Output: SCLK, MOSI, SS (PB3,1,0)
- All other setting are typically determined by peripherals
- Clock rate
  - Determined by datasheets of peripherals
  - Typical clk_{IO} = 16MHz

SPI2X	SPR1	SPR0	SCK	clk_{IO}=16MHz
0	0	0	clk_{IO} /4	4 MHz
0	0	1	clk_{IO} /16	1 MHZ
0	1	0	clk_{IO} /64	250 KHz
0	1	1	clk_{IO} /128	125 KHz
1	0	0	clk_{IO} /2	8 MHz
1	0	1	clk_{IO} /8	2 MHz
1	1	0	clk_{IO} /32	500 KHz
1	1	1	clk_{IO} /64	250 KHz

Coding Example 1

- Configure the MCU SPI as follows:
  - clk_{IO} = 16 MHz:
  - Shift clock of 4-MHz data shift rate
  - Polling method
  - Master mode
  - SCK idle low and data sampled on the rising edge
  - Data shift most significant bit first

```
SPCR = 0x50;     // master, 4MHz SCLK, no interrupt
SPSR = 0;        // clear the SPI2X bit
```

SPI Transmission/Reception

- SPDR = data to be transmitted
  - Reading/writing into SPDR clears flag
- Flag: SPIF = 1 => transmission or reception is complete
  - SPSR = 0x80
- Transmission polling method:

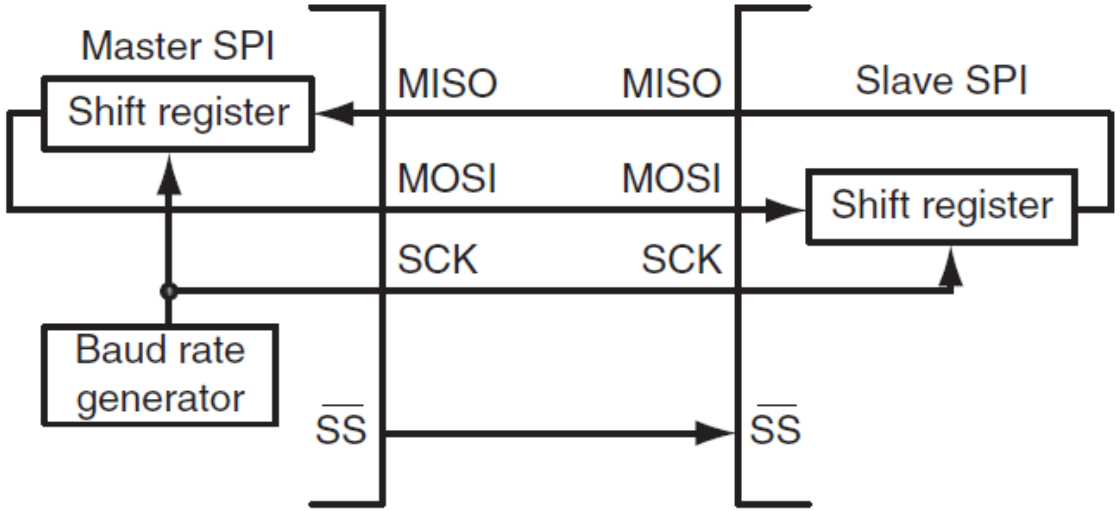
```
SPDR = data;
while(!(SPSR & 0x80)); // wait for data to be shifted out
tmp = SPDR;            // clear SPIF flag
```

- Reception polling method:

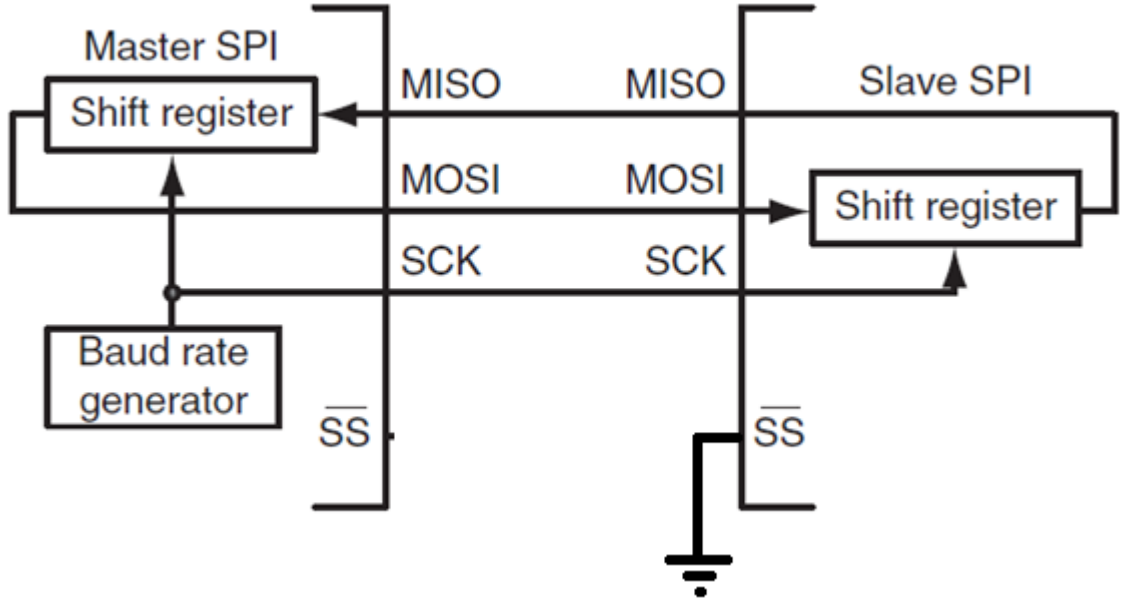
```
while(!(SPSR & 0x80)); // wait for data byte to be shifted in
data = SPDR;           // save data & clear flag
```

Single Peripheral

4-wire



3-wire

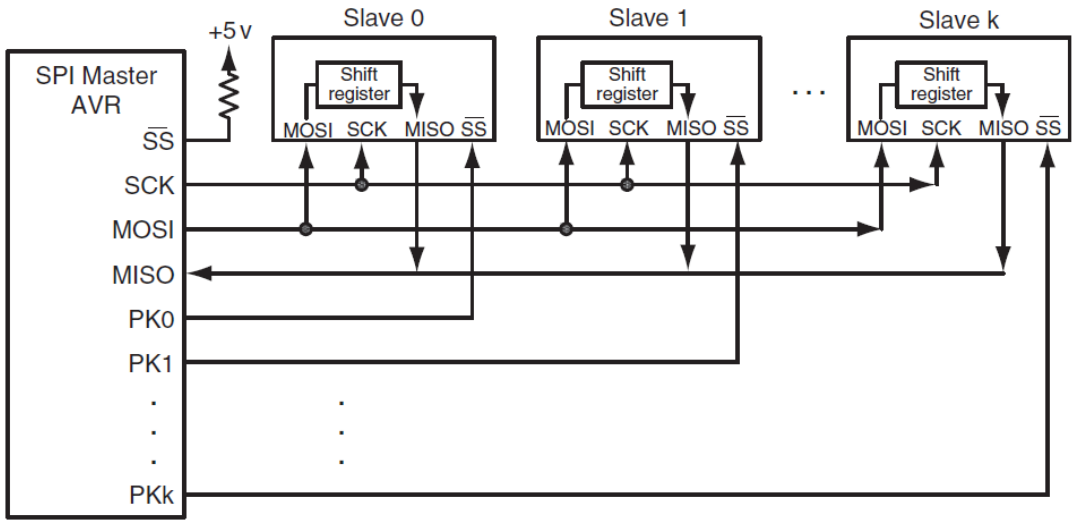


MCU

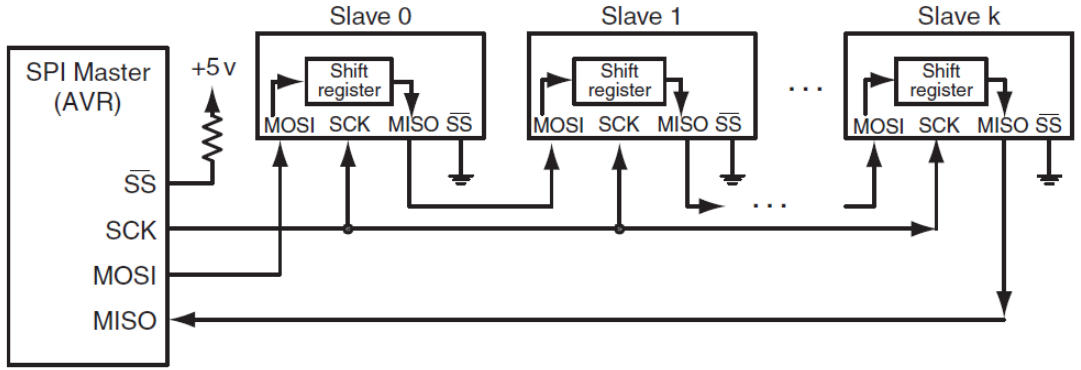
Peripheral

Multiple Peripherals

Parallel/Star

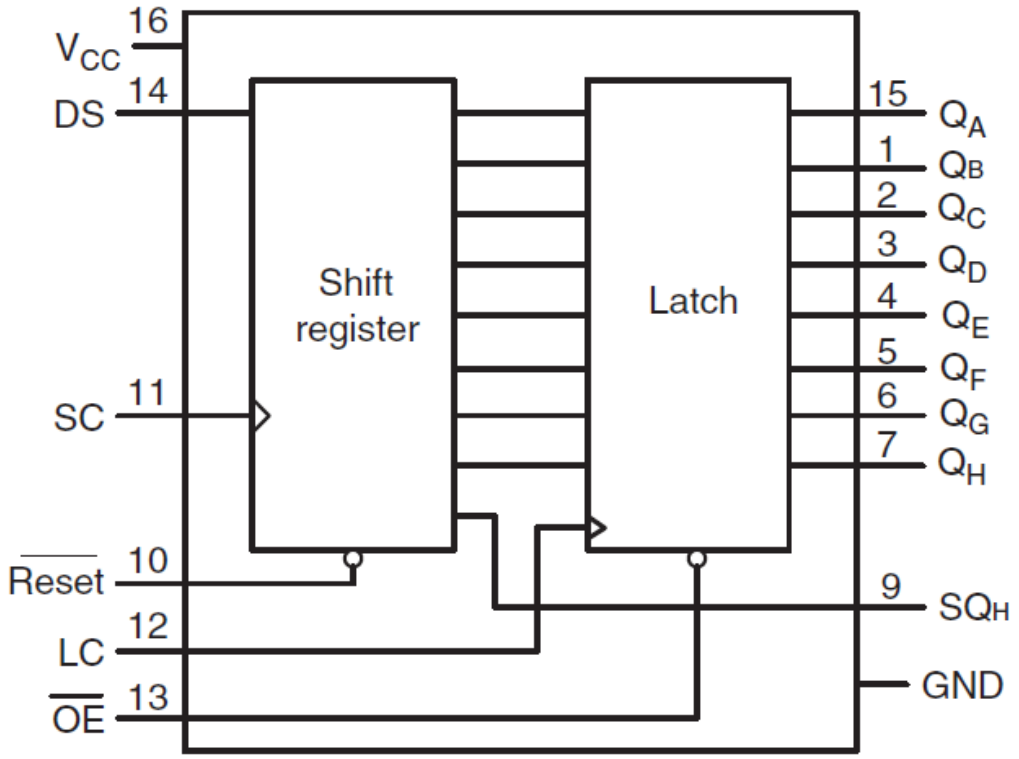


Daisy chained/Ring



HC595 (SIPO) Shift Register

- Converts serial input to parallel output
- 8-bit shift register and a D-type latch with three-state parallel output.
- The maximum data shift rate is 100 MHz (Philips part).
- Pins:
  - **DS**: serial data input
  - **SC**: shift clock. Rising edge triggered
  - **Reset**: low input. Clears shift register.
  - **LC**: latch clock. Rising-edge triggered. Loads shift register to output latch.
  - **OE**: output enable. Low input pin.
  - **QA**: LSB tri-state latch output
  - **QH**: MSB tri-state latch output
  - **SQ_H**: input signal's 8^{th} bit

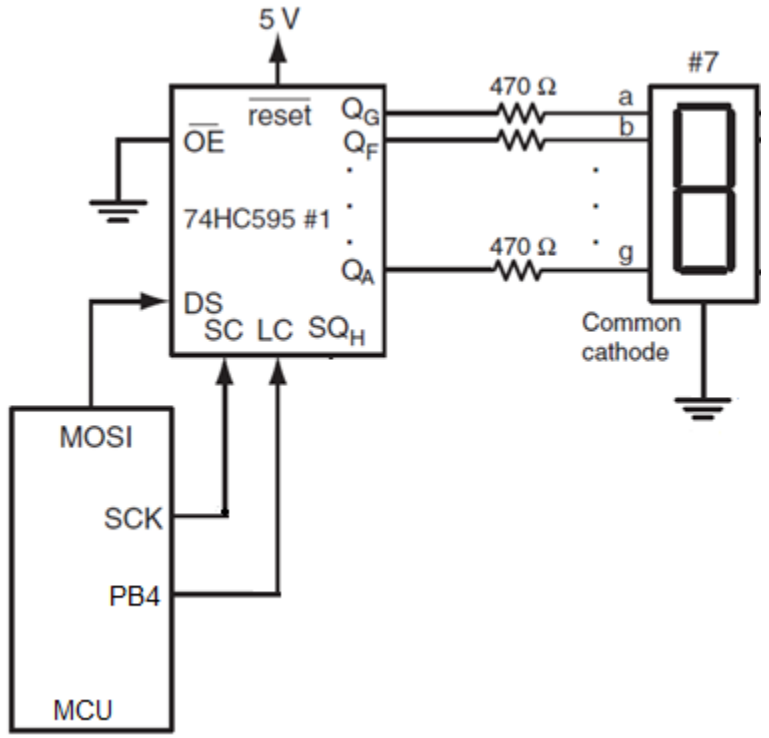


SPI Interfacing with HC595

- Device receives data only
  - MCU is master
- MCU SPI_SCLK → **SC**
  - Rising edge triggered: CPOL, CPHA = 00 or 11
- MCU SPI_MOSI → **DS**
- MCU_PORTxn → **LC**
  - PORTxn is any MCU GPIO pin
  - **LC** = 0 before transmission
  - **LC** = 1 when transmission is complete
- GND → **OE**
  - Always enable tri-state buffer output
  - Use LC to change output
- VCC → **RESET**
  - Never reset

Design Problem 1: Single Shift Register

- 1 Shift registers is used to control a common cathode 7SD's
  - Q_G (MSB) = connected to segment a
  - Q_A (LSB) = connected to segment g
- LC connected to SS
  - This will display the
- Display "0" to "9" on the 7SD in 1 sec intervals



Design Problem 1 Algorithm

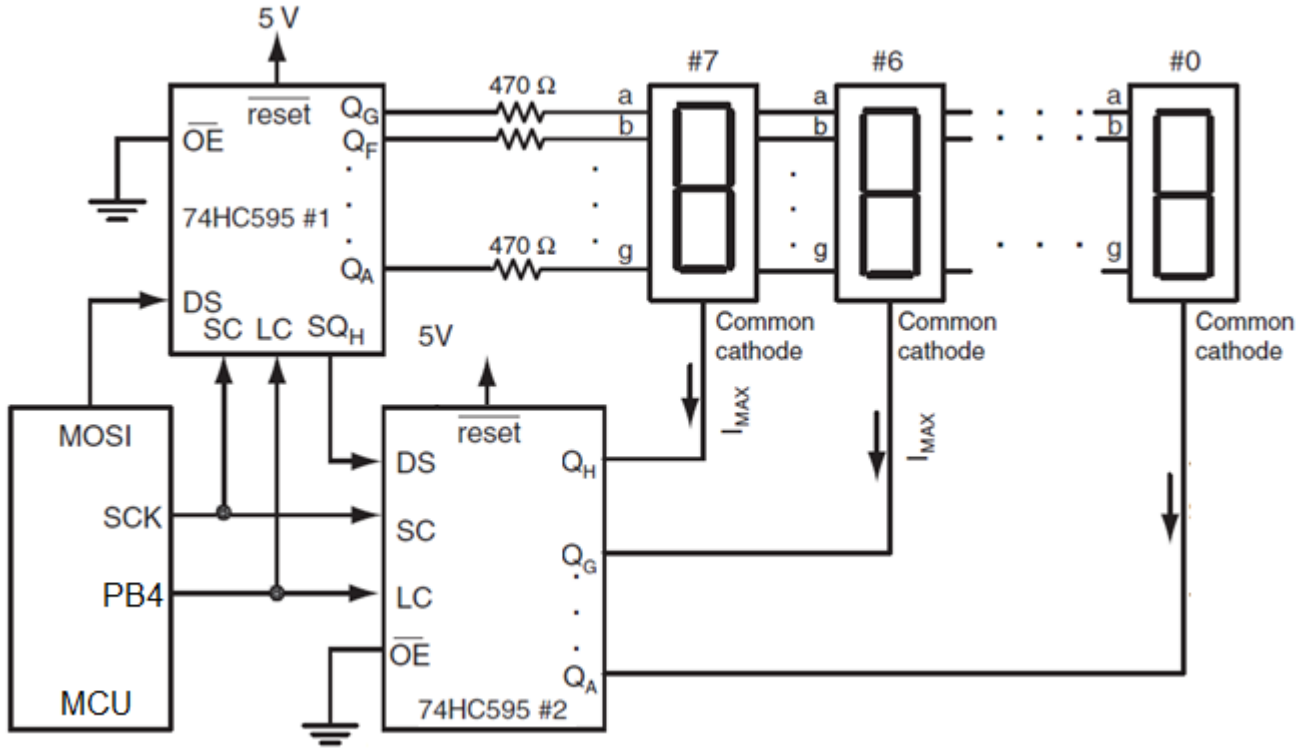
1. Compute array values for 7sd:
   - Q_H (MSB unconnected), seg a $\rightarrow$ Q_G; ... ; seg g $\rightarrow$ Q_A (LSB)
2. Configure SPI
   - MCU master,
   - Clock: 16/4 = 4Mhz , rising edge triggered
   - polling method
   - configure Port B (PB4,3,1,0: output; PB2:input)
3. Transmission protocol:
   - For i = 0 to 9:
     - LC = 0
     - write to SPDR = SSD[i]
     - LC = 1
     - Wait for ~1 sec

Design Problem 1 Code

```
char SSD[]={0x7E, 0x30, 0x6D, 0x79, 0x33, 0x5B, 0x5F, 0x70, 0x7F, 0x7B};

SPCR = 0x50;            // master, 4MHz SCLK, no interrupt
SPSR = 0;               // clear the SPI2X bit

DDRB = 0b00011011;    //PB4=LC;PB3=SCLK;PB2=MISO;PB1=MOSI;PB0=SS

for (int i=0, tmp; i<9; i++)
{       PORTB = 0;                      //LC = 0
        SPDR = SSD[i];                  //transmit SSD pattern
        while(!(SPSR & 0x80));          // wait for data to be shifted out
        tmp = (int) SPDR;               // clear SPIF flag
        PORTB |= 0b00010000;            //LC = 1
        _delay_ms(1000);                //1000ms - 8/4M - 2/16M - 4/16M
}
```

Design Problem 2: Multiple Shift Registers

- 2 Shift registers are used to control 8 common cathode 7SD's
  - 1st shift reg used to hold the pattern to display
    - Q_G (MSB) = connected to segment a
    - Q_A (LSB) = connected to segment g
  - 2nd shift reg used to control which 7SD is ON
    - ON: cathode connected to GND



- Display "12345678" on the 7SD
  - Send 2 bytes
    - 1st byte to select which 7SD is ON
    - 2nd byte to display pattern

Design Problem 2 Algorithm

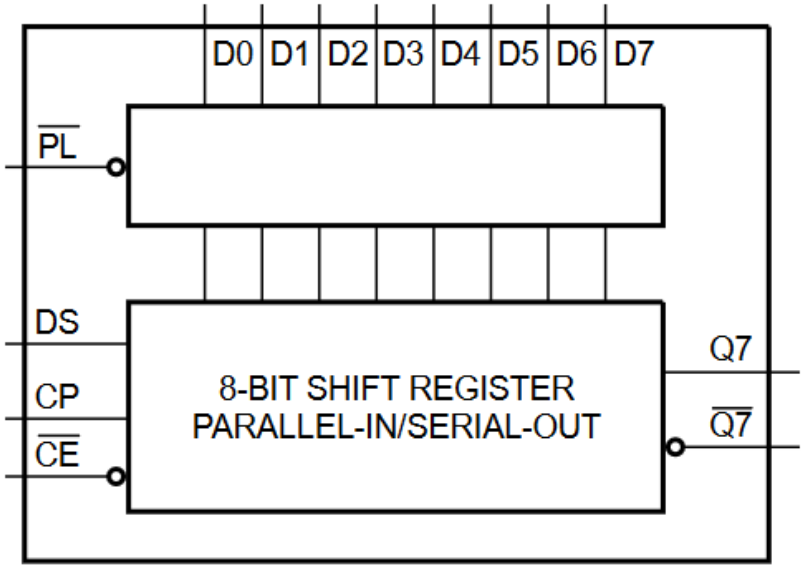
1. Compute array values for 7sd and activation:
   - To activate SSD_7 :GND pin to LO, other $SSD_{6\text{-}1}$ GND pin connected to HI
     - Activation of SSD[7] = ~0x80 = 0b01111111 = 0x7F
     - Activation of SSD[0] = ~0x01 = 0b11111110 = 0xFE
2. Configure SPI
3. Transmission protocol:
   - while(1)
     - For i = 0 to 9:
       - LC = 0
       - Write to SPDR = activation[i]
       - Write to SPDR = SSD[i]
       - LC = 1
       - Wait for 20ms

Design Problem 1 Code

```
char SSD[]={0x7E, 0x30, 0x6D, 0x79, 0x33, 0x5B, 0x5F, 0x70, 0x7F, 0x7B};
char activate[]={~(0x80),~(0x40),~(0x20),~(0x10),~(0x08),~(0x04),~(0x02),~(0x01)};
SPCR = 0x50;                  // master, 4MHz SCLK, no interrupt
SPSR = 0;                     // clear the SPI2X bit
DDRB = 0b00011011;            //PB4=LC;PB3=SCLK;PB2=MISO;PB1=MOSI;PB0=SS
while(1)
{   for (int i=0; i<9; i++)
    {       PORTB &= 0b11101111;          //LC = 0
            putcSPI(activate[i]);         //activate 7sd i
            putcSPI(SSD[i]);              //send pattern
            PORTB |= 0b00010000;          //LC = 1
            _delay_ms(20);                //
    }
}
void putcSPI(char var8)
{   int tmp;
    SPDR = var8;                  //transmit SSD pattern
    while(!(SPSR & 0x80));        // wait for data to be shifted out
    tmp = SPDR;                   // clear SPIF flag
}
```

HC165 (PISO) Shift Register

- 8-bit shift register
- Parallel-in, serial-out shift register
- Typical frequency for 5V Vcc is 50MHz
  - https://assets.nexperia.com/documents/data-sheet/74HC_HCT165.pdf
- Pins:
  - **D0-D7**: Parallel data in
  - **PL**: parallel load input (active LOW)
  - **CP**: clock input rising edge triggered
  - **CE**: clock enable input (active LOW)
  - **Q7**: serial output from the last stage
    - Complementary Q7
  - **DS**: serial data input

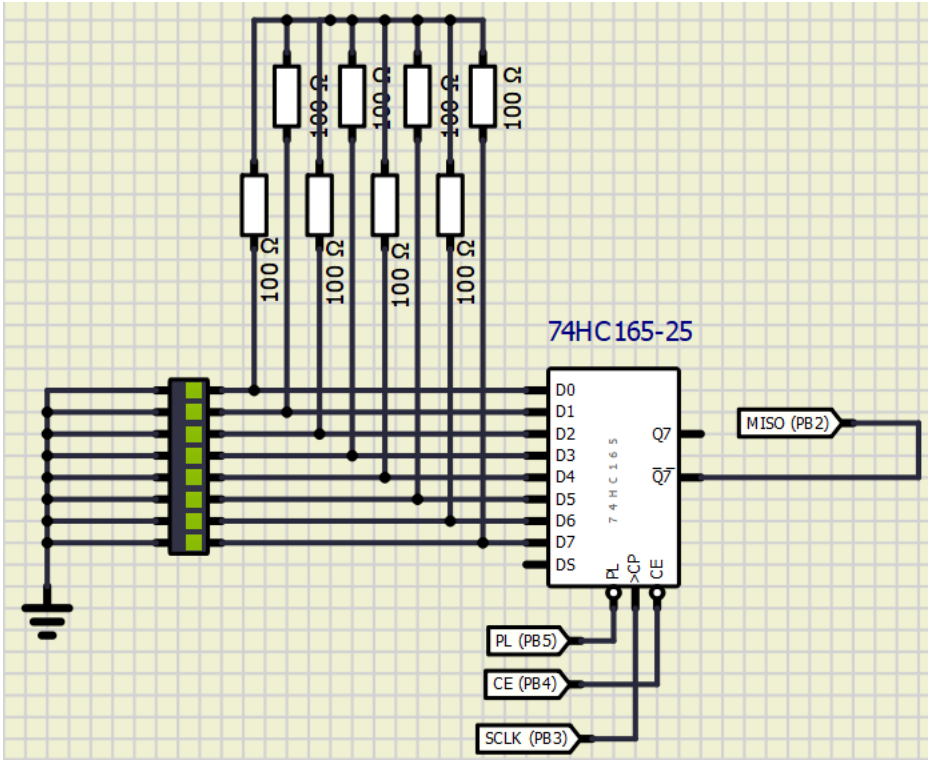


SPI Interfacing with HC165

- MCU supplied clock: master
  - MCU SPI_SCLK → **CP**
  - Rising edge triggered: CPOL, CPHA = 00 or 11
- MCU receives data only
  - **Q7** → MCU SPI_MISO
- MCU_PORTxn → **PL**
  - PORTxn is any MCU GPIO pin
  - **PL** = 0 to load parallel input before transmission
  - **PL** = 1 to stop loading input
- **CE** = 0 to enable shifting
  - **CE** = 1 while loading
- **CE** and **PL** act complimentary
  - When **PL**=0 (loading) then **CL** = 1 (can't shift)
  - When **CL**=0 (shifting out) then **PL** = 1 (can't load)

Design Problem 4: Serial DIPS

- 8x DIPS: PULL-UP when OFF
  - ON = GND => must complement
- MCU SPI:
  - Master
  - MISO: connected to Q7 complement
  - SCLK: conntected to CP
- MCU PORT pins
  - PB5: used to load enable
  - PB4: used for clock enable



Design Problem 4: Algorithm

1. Setup the MCU
   - SPI as master
   - PortB, pins 4,5 as output
2. Read DIPS:
   - Load the DIPS input into the shift register: PE = 0, CE = 1 (load, no shifting)
   - Enable clock and disable load (PE=1, CE=0)
   - Read the SPIDR

Design Problem 4 Code

```
char data;

SPCR = 0x50;              // master, 4MHz SCLK, no interrupt
SPSR = 0;                 // clear the SPI2X bit
DDRB = 0b00111011;        //PB5=PL;PB4=CE;PB3=SCLK;PB2=MISO;PB1=MOSI;PB0=SS

PORTB |= 0b00010000;      //CE=1 (no shifting)
PORTB &= 0b11011111;      //PL=0 (load DIPS)
//_delay_1ms(1);          //wait for input
PORTB |= 0b00100000;      //PL=1 (stop loading)
PORTB &= 0b11101111;      //CE=0 (start shifting)

while(!(SPSR & 0x80)); // wait for data byte to be shifted in
data = SPDR;              // save data & clear flag
```

USART in SPI Mode

- 1xSPI vs 4xUSART
- USART in MSPI mode
  - UCSRnC reg, UMSELn10 bits = 11 => UCSRnC |= 0b11000000
  - TxD is MOSI, TxD is MISO, XCLK is SCK
    - SS is not is MSPI
  - Frame can only be 8-bit
  - Interrupt timing not compatible with SPI
  - WCOL SPI flag not in MSPI
  - UCSRnC reg
    - UCPOLn bit = 0 (rising edge), CPHA,CPOL = 00, 11
    - UCPOLn bit = 1 (falling edge), CPHA,CPOL = 01, 10
  - MSPI clock (baud) based on $\mathrm{UBBR}\mathbf{n} = \frac{\mathrm{f_{clkIO}}}{2\times\mathrm{Baud}} - 1$
    - baud rate register must be zero at the time the transmitter is enabled.

MSPI Setup Example

- Configure USART0 operate in MSPI mode with the following specifications:
  - Enable transmitter and receiver
  - XCK idle low, shift data on the rising edge
  - Baud rate of 8MHz

```
UBBR0  = 0;      //must be 0 at transmission enable
DDRE = 0x06;     //TXD, RXD pin enable
UCSR0B = 0x18;   //enable transmitter and receiver
UCSR0C = 0xC0;   // MSPI mode, XCK idle low, rising edge
UBRR0  = 0;      //16M/(2*8)-1=0 baud rate
```

MSPI Tx/Rx

- Data Tx:

```
while(!(UCSR0A & 0x20));      // wait until UDR is empty
UDR0 = data;                  // sends the character
while(!(UCSR0A & 0x40));      // wait until data is shifted out
tmp  = UDR0;                  // clear the RXC0 flag
```

- Data Rx:

```
while(!(UCSR0A & 0x20));      // wait until UDR is empty
UDR0 = 0;                     // trigger clock signals from XCK0 pin
while(!(UCSR0A & 0x80));      // wait until data is shifted in
tmp  = UDR0;                  // clear the RXC0 flag
```