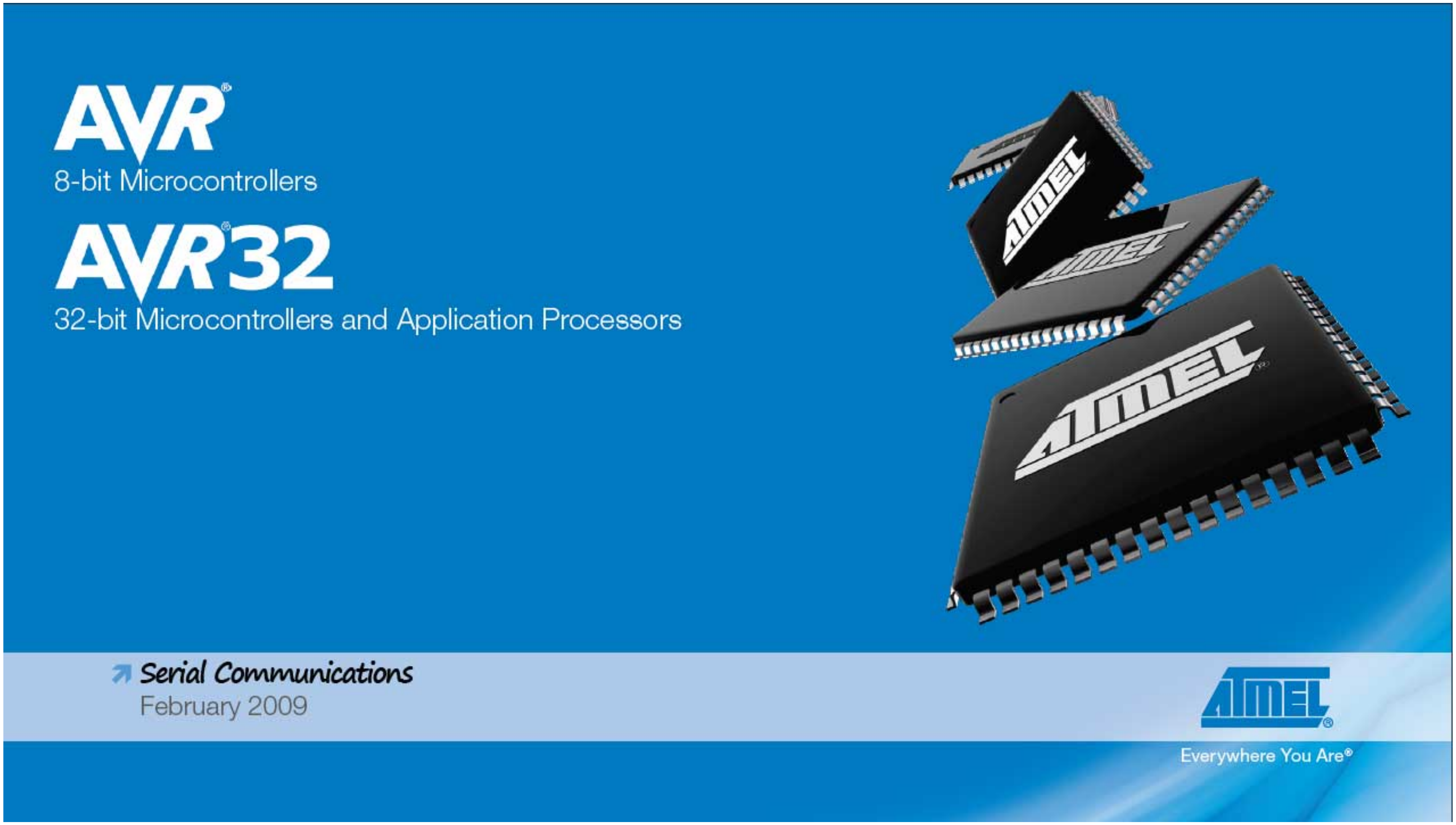


AVR®

8-bit Microcontrollers

AVR®32

32-bit Microcontrollers and Application Processors

Serial Communications

February 2009

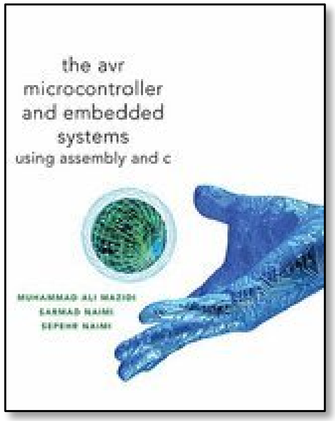
ATMEL®

Everywhere You Are®

Serial Communications - SPI

READING

The AVR Microcontroller and Embedded Systems using Assembly and C)
by Muhammad Ali Mazidi, Sarmad Naimi, and Sepehr Naimi



Chapter 5: Arithmetic, Logic Instructions, and Programs

Section 5.4: Rotate and Shift Instructions and Data Serialization

Chapter 7: AVR Programming in C

Section 7.5 Data Serialization in C

Chapter 11: AVR Serial Port Programming in Assembly and C

Section 11.1 Basics of Serial Communications only (*You are not responsible for Sections 11.2 to 11.5*)

Chapter 17: SPI Protocol and MAX7221 Display Interfacing

Section 17.1 SPI Bus Protocol

Additional Resource Material

- Fairchild Semiconductor MM74HC595 “8-Bit Shift Register with Output Latches” document MM74HC595.pdf
- Arduino Wire Library http://www.arduino.cc/en/Reference/Wire
- Arduino Interfacing with Hardware http://www.arduino.cc/playground/Main/InterfacingWithHardware

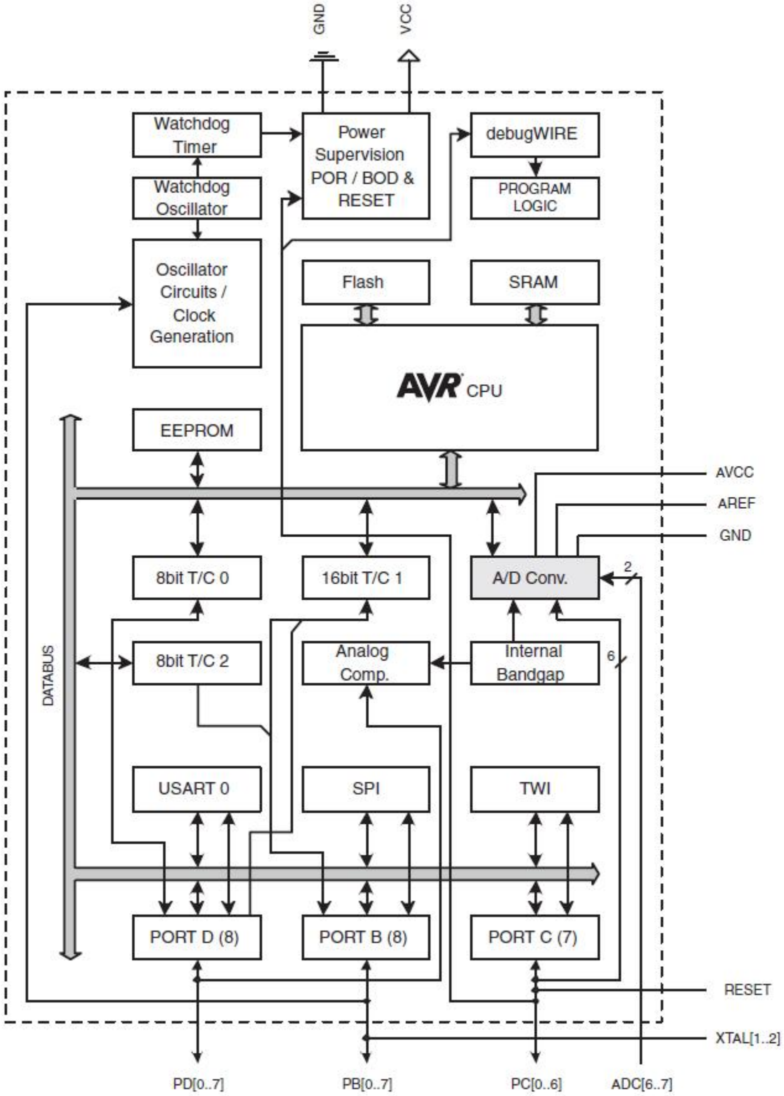
Location of Arduino Wire Library C: \Program Files (x86)\ arduino-0017\ hardware\ libraries

Location of #include files stdlib.h, string.h, inttypes.h C: \Program Files (x86)\arduino-0017\ hardware\tools\avr\avr\include

Location of #include file twi.h (1 of 3) C: \Program Files (x86)\arduino-0017\ hardware\tools\avr\avr\include\compat

TABLE OF CONTENTS

ATMEGA328P BLOCK DIAGRAM[1]



[1] Source: ATmega328P Data Sheet http://www.atmel.com/dyn/resources/prod_documents/8161S.pdf page 5

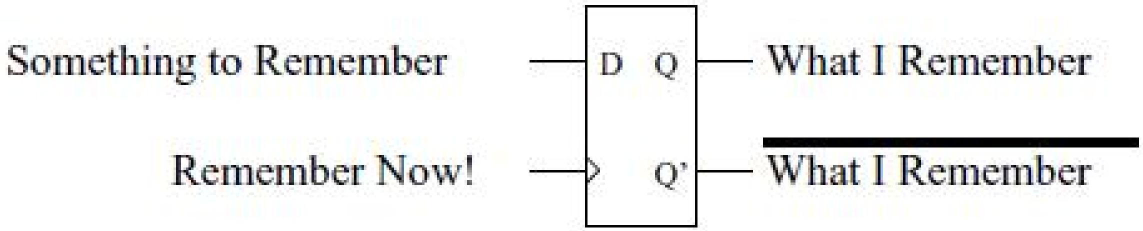
ATmega SPI – Serial Peripheral Interface

- 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.

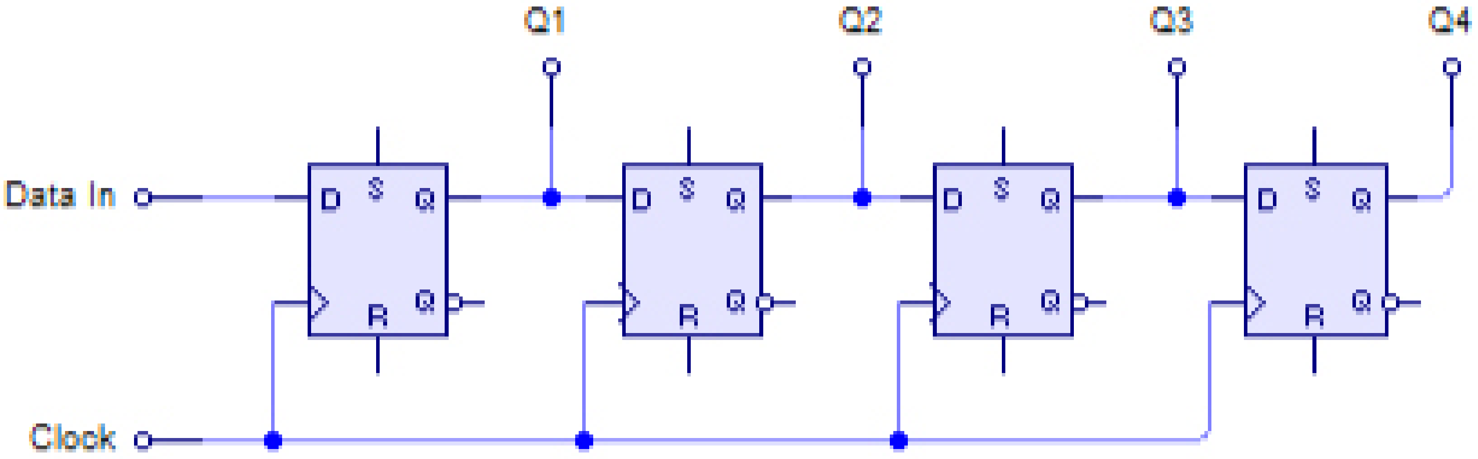
What is a Flip-Flop and a Shift Register

You can think of a D flip-flop as a one-bit memory. The *something to remember* on the D input of flip-flop is remembered on the positive edge of the clock input.

D_t	Q_{t+1}
0	0
1	1
X	Q_t

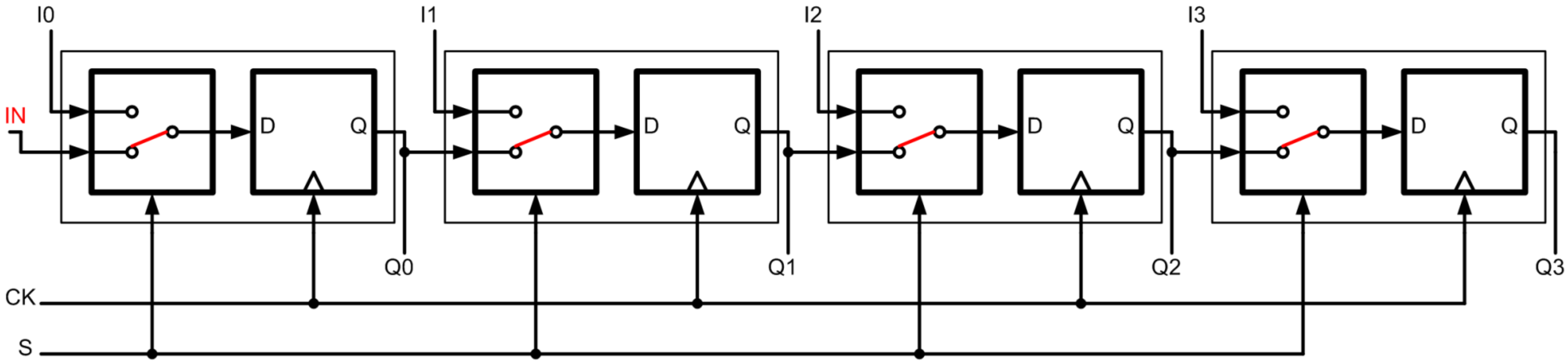


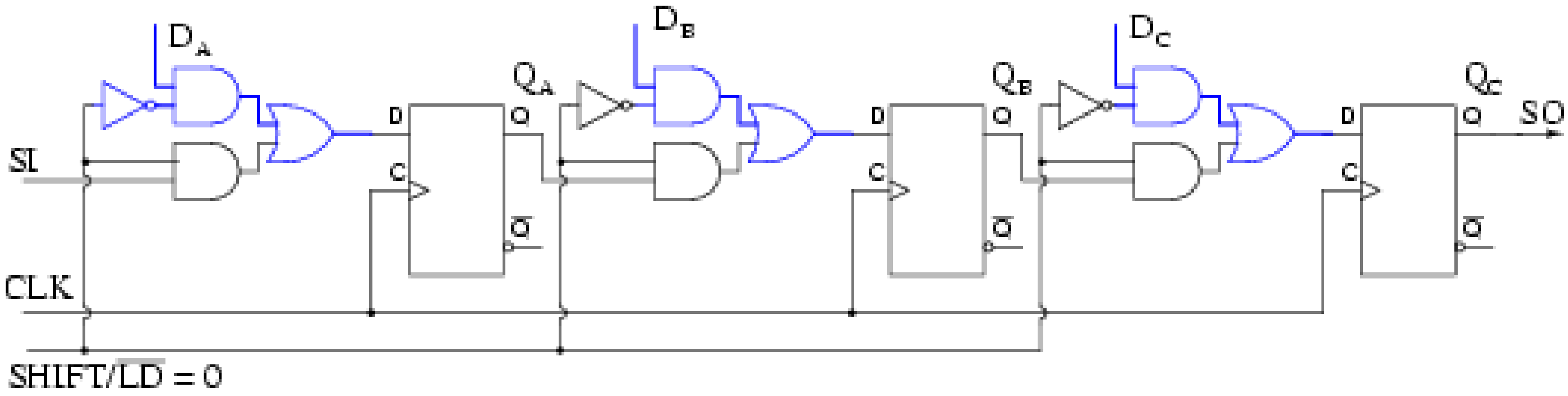
A data string is presented at 'Data In', and is shifted right one stage on each positive 'Clock' transition. At each shift, the bit on the far left (i.e. 'Data In') is shifted into the first flip-flop's output (i.e., ‘Q’).[2]



[2] Source: http://en.wikipedia.org/wiki/File:4-Bit_SIPO_Shift_Register.png

WHAT IS A SERIAL SHIFT REGISTER WITH PARALLEL LOAD





Parallel-in/ serial-out shift register showing parallel load path

Nine Questions that need answers before you can design a Serial Peripheral Interface

Configuration and Control

1.	Mstr: Master/Slave Select	Who is the Master and who is the Slave? Specifically, which subsystem contains the clock?
2.	SPI Clock Rate Select	At what clock frequency (divisor) is the data transmitted/received by the Master?
3.	Data Order	In what order is the data transmitted (msb or lsb first)?
4.	Clock Polarity & Phase	How is the data transmitted relative to the clock (data setup and data sampled)
5.	SPI Enable	How do you enable the clock on and off?
6.	SPI Interrupt Enable	How do you enable/disable the interrupt flag?

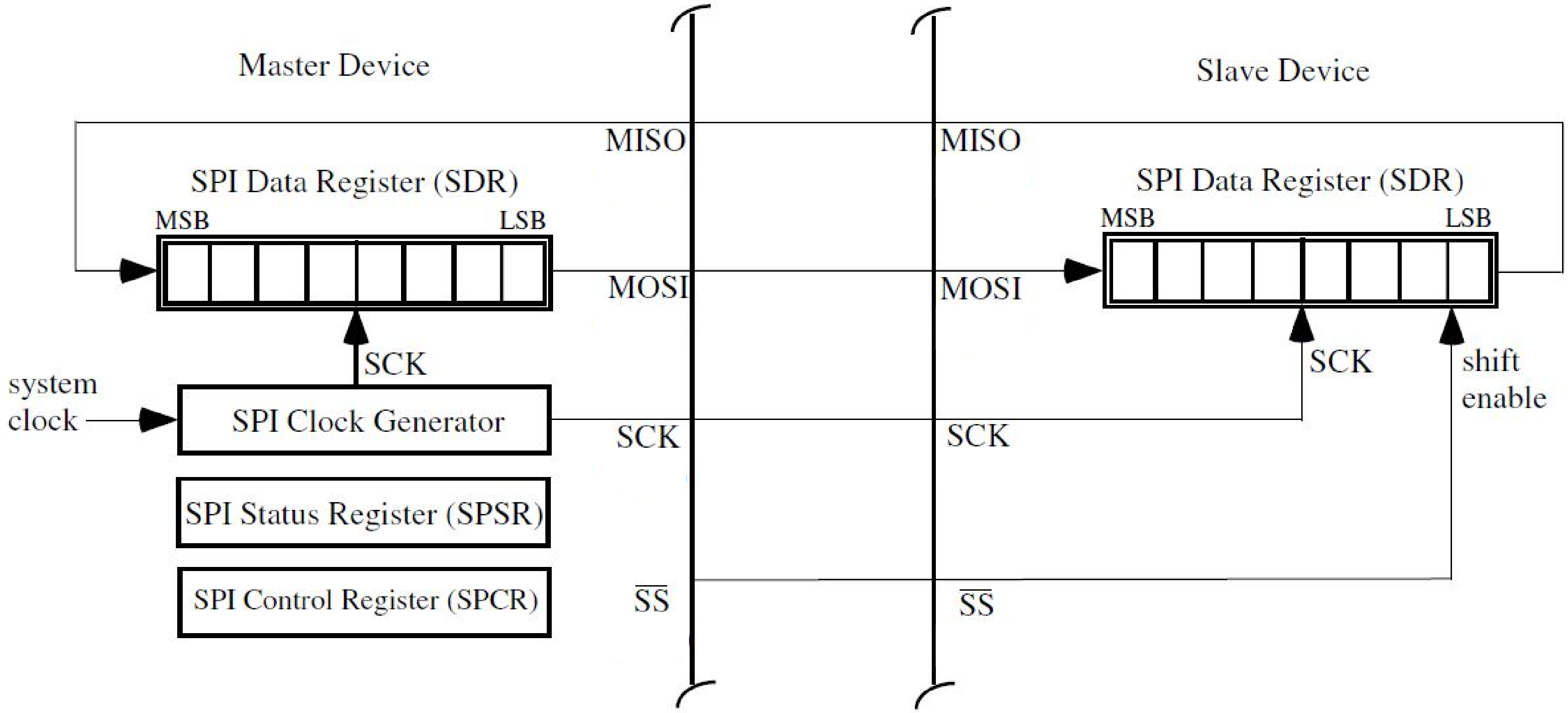
Send/Receive Data

7.	SPDR Write	How do you write data to the SPI Data Register?
8.	SPDR Read	How do you read data to the SPI Data Register?

Monitoring and Status Questions

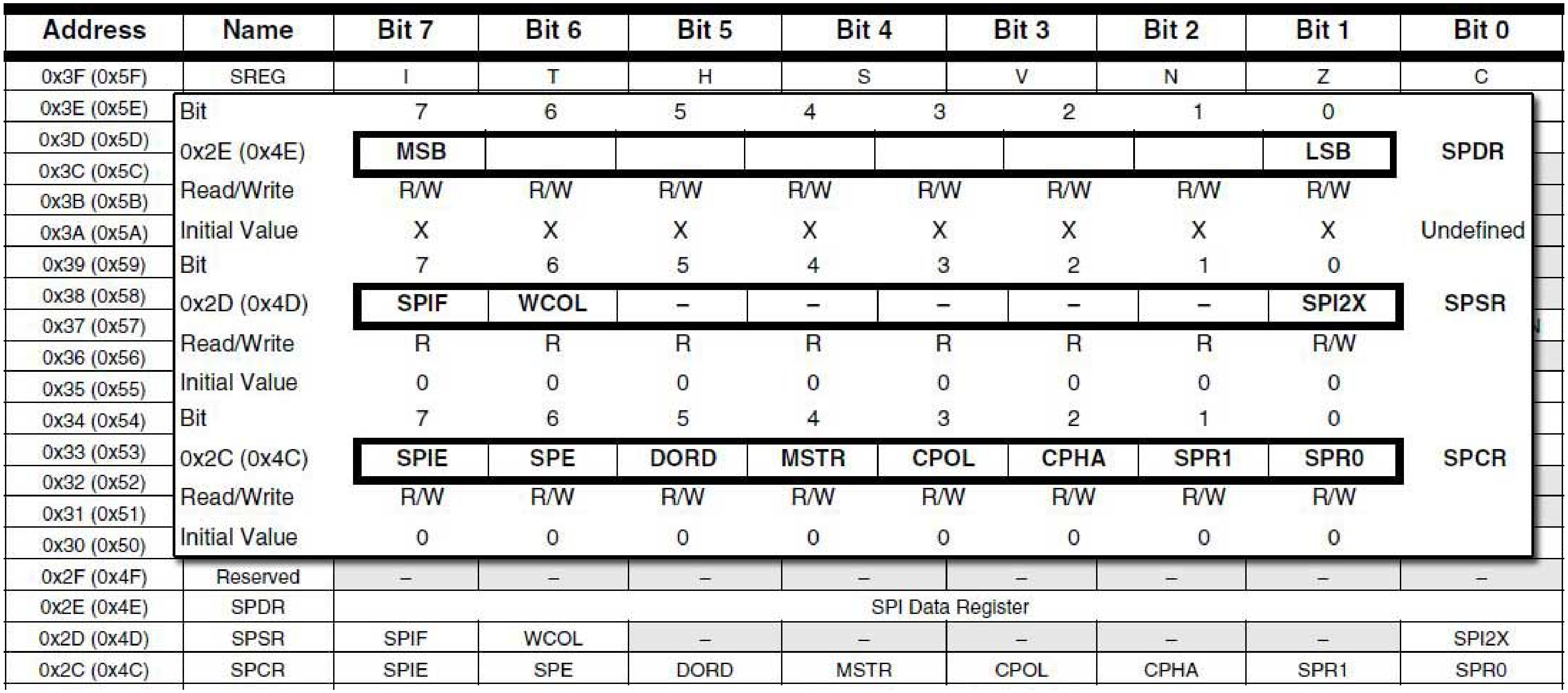
9.	SPI Interrupt Flag	How do you know when a data transfer operation is done?
10.	Write Collision Flag	How do you detect if a byte of data was written to the shift register during a data transfer operation.

SPI Overview – Serial Communication



- *SPI Control Register* – You configure the SPI subsystem by writing to the SPI Control Register (**SPCR**) and the SPI2X bit of register SPSR. The ATmega328P SPI subsystem may be configured as a *master* a *slave* or both. Setting bit SPE bit enables the SPI subsystem.
- *SPI Data Register* – Once enabled (SPE = 1), writing to the SPI Data Register (**SPDR**) begins SPI transfer.
- *SPI Status Register* – The **SPSR** register contains the SPIF flag. The flag is set when 8 data bits have been transferred from the master to the slave. The WCOL flag is set if the SPI Data Register (SPDR) is written during the data transfer process.

SPI OVERVIEW – THE REGISTERS



Address	Name	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
0x3F (0x5F)	SREG	I	T	H	S	V	N	Z	C
0x3E (0x5E)									
0x3D (0x5D)									
0x3C (0x5C)									
0x3B (0x5B)									
0x3A (0x5A)									
0x39 (0x59)									
0x38 (0x58)									
0x37 (0x57)									
0x36 (0x56)									
0x35 (0x55)									
0x34 (0x54)									
0x33 (0x53)									
0x32 (0x52)									
0x31 (0x51)									
0x30 (0x50)									
0x2F (0x4F)	Reserved	–	–	–	–	–	–	–	–
0x2E (0x4E)	SPDR	SPI Data Register							
0x2D (0x4D)	SPSR	SPIF	WCOL	–	–	–	–	–	SPI2X
0x2C (0x4C)	SPCR	SPIE	SPE	DORD	MSTR	CPOL	CPHA	SPR1	SPR0

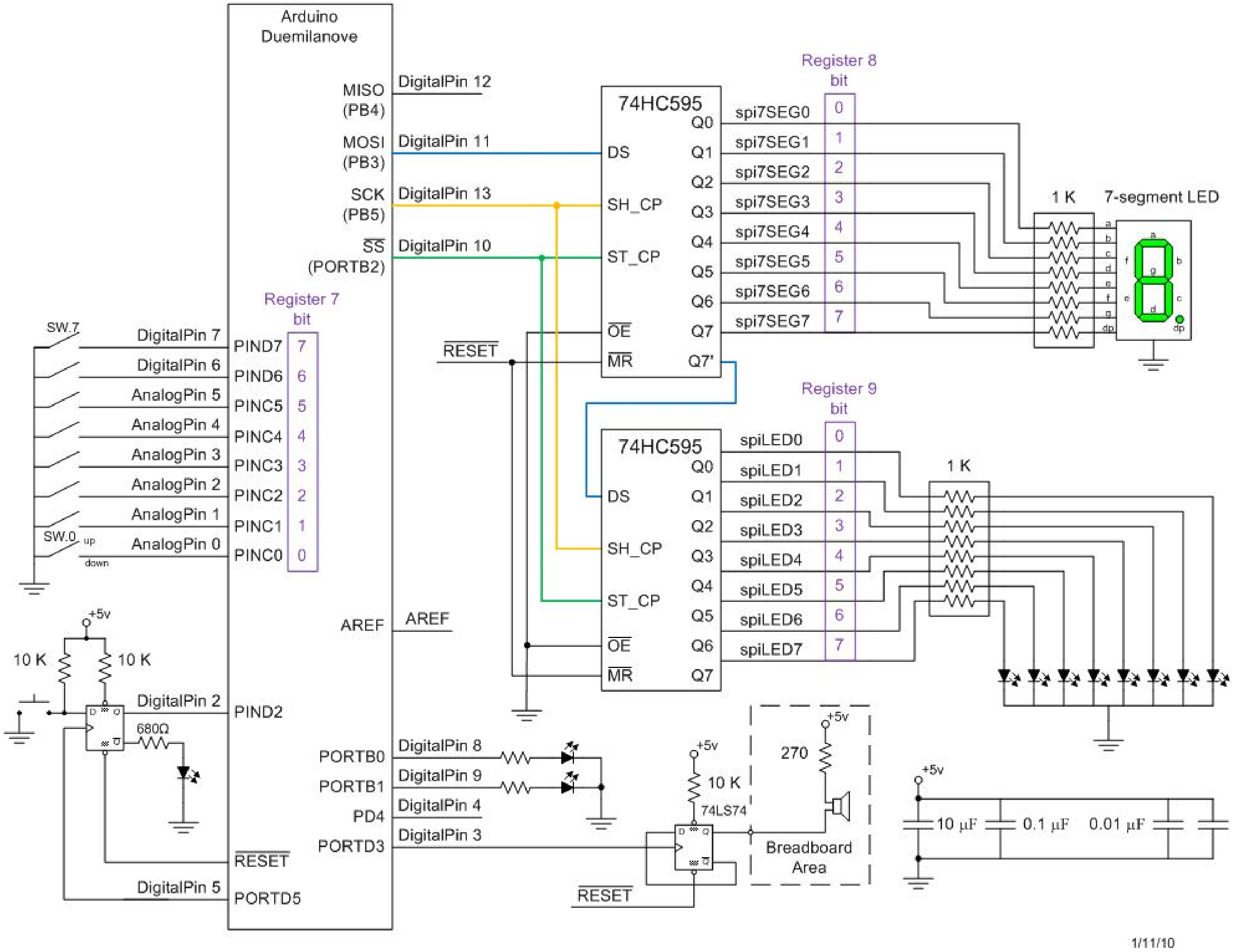
Bit	7	6	5	4	3	2	1	0	
0x2E (0x4E)	MSB							LSB	SPDR
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
Initial Value	X	X	X	X	X	X	X	X	Undefined

Bit	7	6	5	4	3	2	1	0	
0x2D (0x4D)	SPIF	WCOL	–	–	–	–	–	SPI2X	SPSR
Read/Write	R	R	R	R	R	R	R	R/W	
Initial Value	0	0	0	0	0	0	0	0	

Bit	7	6	5	4	3	2	1	0	
0x2C (0x4C)	SPIE	SPE	DORD	MSTR	CPOL	CPHA	SPR1	SPR0	SPCR
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

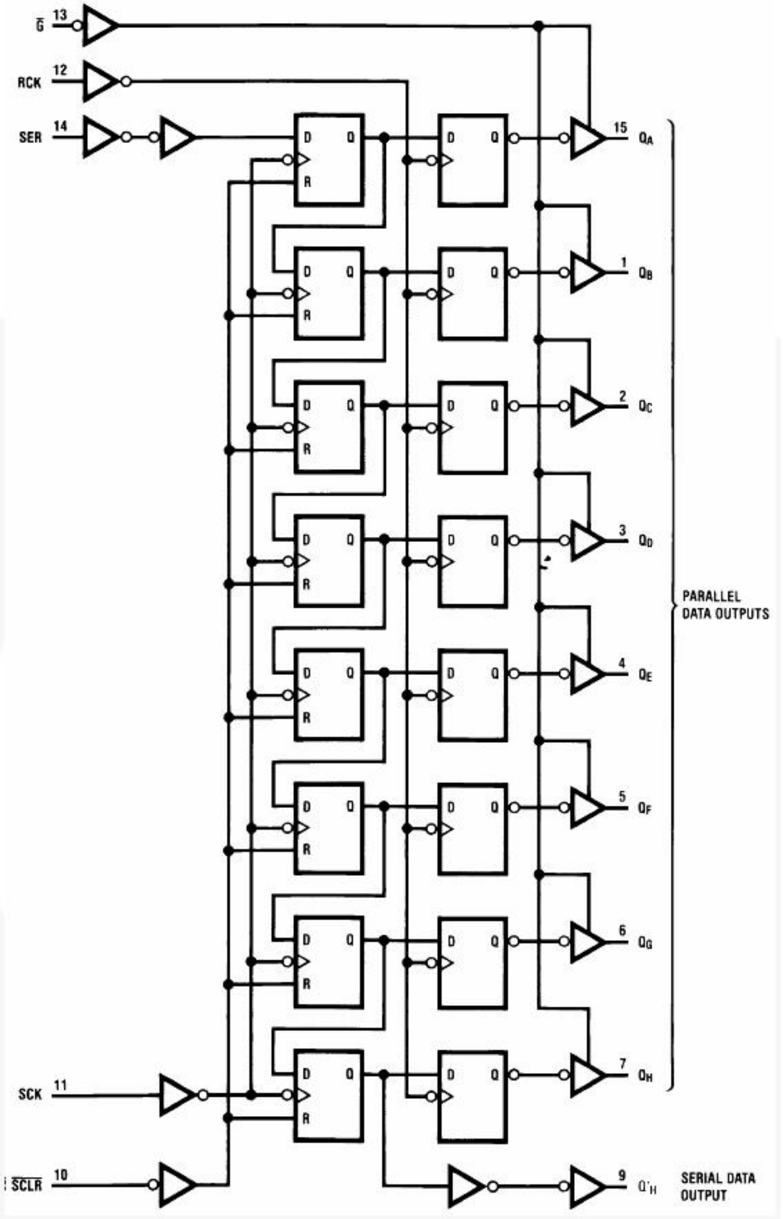
- *SPI Control Register* – You configure the SPI subsystem by writing to the SPI Control Register (**SPCR**) and the SPI2X bit of register SPSR. The ATmega328P SPI subsystem may be configured as a *master* a *slave* or both. Setting bit SPE bit enables the SPI subsystem.
- *SPI Data Register* – Once enabled (SPE = 1), writing to the SPI Data Register (**SPDR**) begins SPI transfer.
- *SPI Status Register* – The **SPSR** register contains the SPIF flag. The flag is set when 8 data bits have been transferred from the master to the slave. The WCOL flag is set if the SPI Data Register (SPDR) is written during the data transfer process.

SPI Design Example – Arduino Proto-Shield

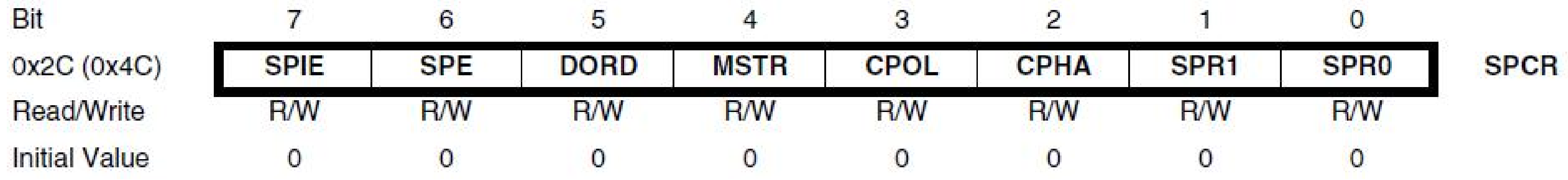


OVERVIEW OF THE 74HC595

- The 74HC595 “8-bit Shift Register with Output Latches” contains an eight-bit serial-in (**SER**), parallel-out (**Q_H to Q_A**), shift register that feeds an eight-bit D-type storage register.
- The storage register has eight 3-state outputs, controlled by input line **G**.
- Separate **positive-edge triggered** clocks are provided for both the shift register (**SCK**) and the storage register (**RCK**)
- The shift register has a direct overriding clear (**SCLR**), serial input, and serial output (standard) pins for cascading (**Q'_H**).



How to Configure the SPI Subsystem



Bit	7	6	5	4	3	2	1	0	
0x2C (0x4C)	SPIE	SPE	DORD	MSTR	CPOL	CPHA	SPR1	SPR0	SPCR
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

SPI Interrupt Enable = 0

This bit causes the SPI interrupt to be executed if the SPIF bit in the SPSR Register is set and if the Global Interrupt Enable bit in SREG is set. For our design example we will be polling the SPIF bit. Consequently, we will leave the SPIE bit in its default (SPIE = 0) state.

SPI Enable = 1

When the SPE bit is one, the SPI is enabled. This bit must be set to enable any SPI operations.

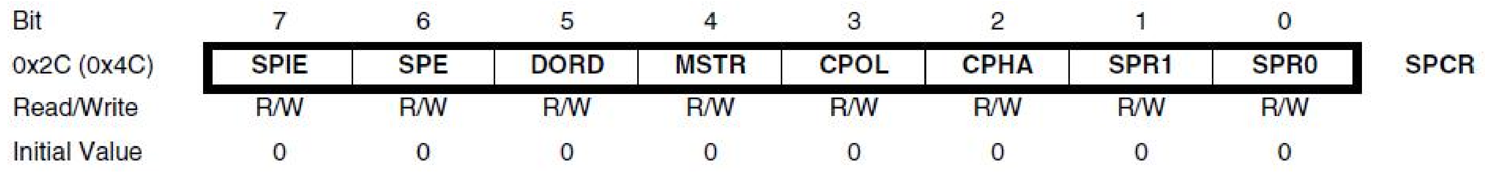
Data Order = 0

When the DORD bit is one (DORD = 1), the LSB of the data word is transmitted first, otherwise the MSB of the data word is transmitted first. For the Arduino Proto-shield, we want to transfer the most significant bit (MSB) bit first. Consequently, we will leave the DORD bit in its default (DORD = 0) state.

MSTR: Master/Slave Select = 1

This bit selects Master SPI mode when set to one, and Slave SPI mode when cleared. For our design example, the ATmega328P is the master and the74HC595 “8-bit Shift Register with Output Latches” is the slave. Consequently, we need to set the DORD bit to logic 1 (MSTR = 1). Note: *I am only telling you part of the story. If you want to configure the ATmega328 to operate as a slave or master/slave please see the datasheet.*

HOW TO CONFIGURE THE SPI SUBSYSTEM – CONTINUED –

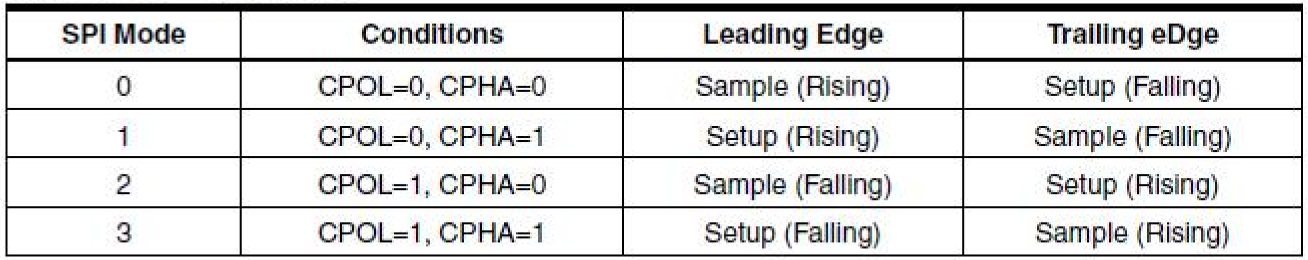


Bit	7	6	5	4	3	2	1	0	
0x2C (0x4C)	SPIE	SPE	DORD	MSTR	CPOL	CPHA	SPR1	SPR0	SPCR
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

Clock Polarity = 0 and Clock Phase = 0

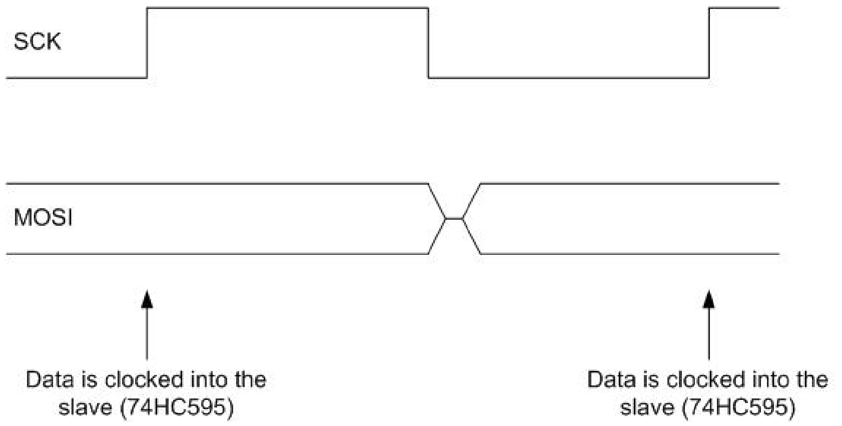
The Clock Polarity (CPOL) and Clock Phase (CPHA) bits define how serial data is transferred between the master and the slave. These SPI Data Transfer Formats are defined in Table 18-2.

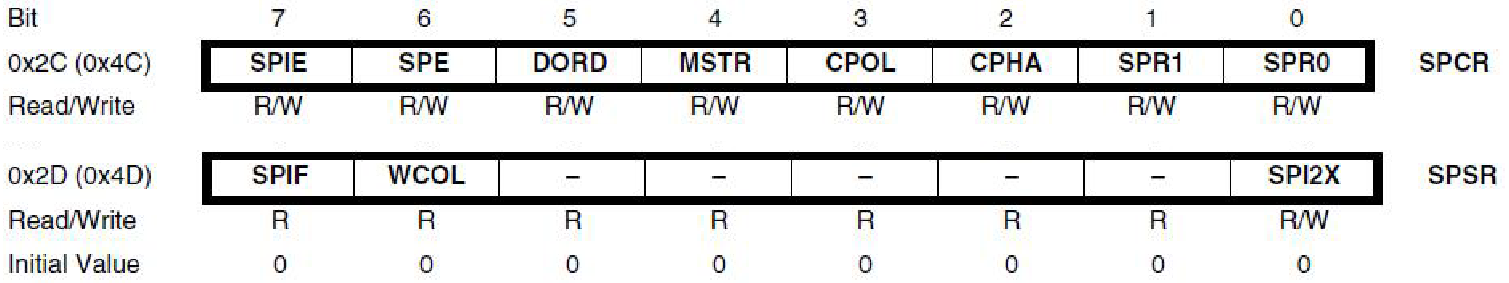
Table 18-2. SPI Modes



SPI Mode	Conditions	Leading Edge	Trailing eDge
0	CPOL=0, CPHA=0	Sample (Rising)	Setup (Falling)
1	CPOL=0, CPHA=1	Setup (Rising)	Sample (Falling)
2	CPOL=1, CPHA=0	Sample (Falling)	Setup (Rising)
3	CPOL=1, CPHA=1	Setup (Falling)	Sample (Rising)

For our design example, we want data to be clocked into the 74HC595 on the Rising clock edge (the D-flip-flops of the 74HC595 are positive edge triggered), as shown in the Figure below. Consequently, we want the ATmega328P to Setup the data on the serial data out line (SER) on the Trailing clock edge. Looking at Table 18-2 we see that this corresponds to SPI Mode = 0.


SCK
MOSI
Data is clocked into the slave (74HC595)
Data is clocked into the slave (74HC595)

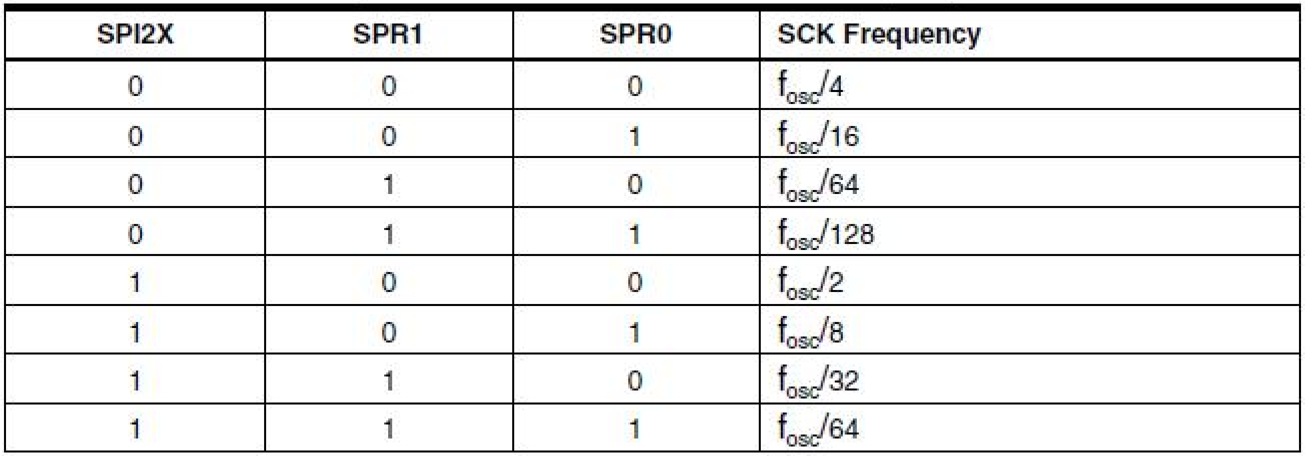


Bit	7	6	5	4	3	2	1	0	
0x2C (0x4C)	SPIE	SPE	DORD	MSTR	CPOL	CPHA	SPR1	SPR0	SPCR
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
0x2D (0x4D)	SPIF	WCOL	–	–	–	–	–	SPI2X	SPSR
Read/Write	R	R	R	R	R	R	R	R/W	
Initial Value	0	0	0	0	0	0	0	0	

SPI Clock Rate Select Bits SPI2X , SPR1, SPR0 = 001_2

These three bits control the SCK rate of the Master. In our design example, the ATmega328P is the Master. These bits have no effect on the Slave. The relationship between SCK and the Oscillator Clock frequency f_{osc} is shown in the following table. For our design example we will be dividing the system clock by 16.

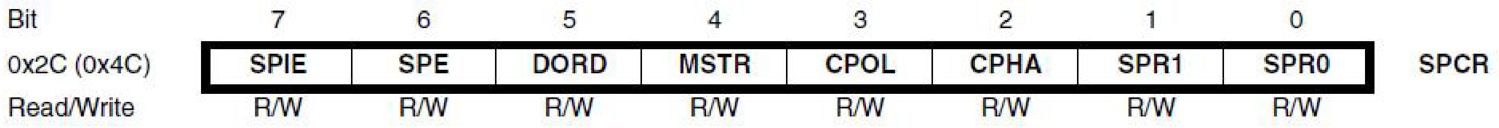
Table 18-5. Relationship Between SCK and the Oscillator Frequency



SPI2X	SPR1	SPR0	SCK Frequency
0	0	0	$f_{osc}/4$
0	0	1	$f_{osc}/16$
0	1	0	$f_{osc}/64$
0	1	1	$f_{osc}/128$
1	0	0	$f_{osc}/2$
1	0	1	$f_{osc}/8$
1	1	0	$f_{osc}/32$
1	1	1	$f_{osc}/64$

HOW TO CONFIGURE THE SPI SUBSYSTEM – CONTINUED –

Reviewing the last three pages, to configure the SPI subsystem for the EE346 Protoshield we need to…



Bit	7	6	5	4	3	2	1	0	
0x2C (0x4C)	SPIE	SPE	DORD	MSTR	CPOL	CPHA	SPR1	SPR0	SPCR
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	

0 1 0 1 0 0 0 1

- We will be polling the SPI Interrupt flag (SPIE)
- Enable the SPI Subsystem (SPE)
- Set Data Order to transmit the MSB first (DORD)
- Define ATmega328P as the Master (MSTR)
- Configure the SPI to clock data (sample) on the rising edge and change data (setup) on the falling edge.
- Set prescalar to divide system clock by 16

C Code Example

```
/* Configure SPI Control Register */
SPCR = (1<<SPE)|(1<<MSTR)|(1<<SPR0);
```

Assembly Code Example

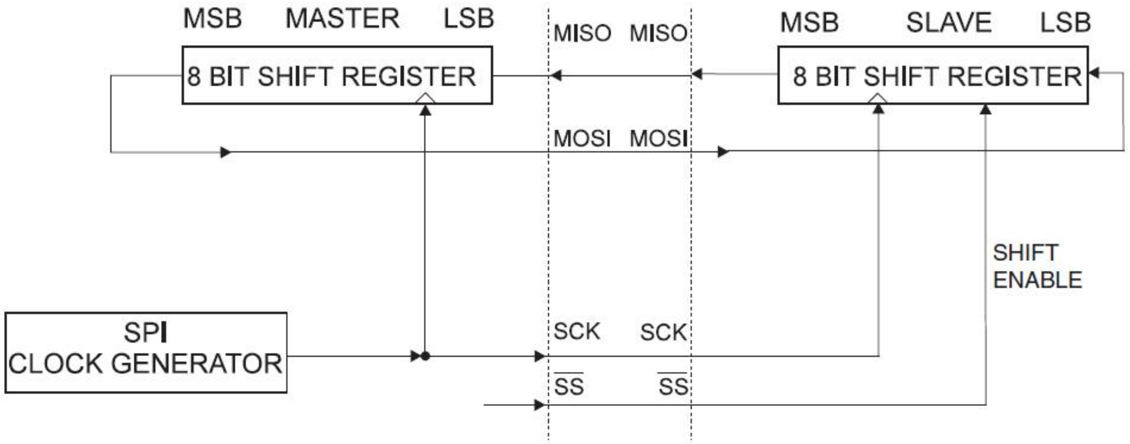
```
ldi  r16, 0x51
out  SPCR, r16
```

Test Your Knowledge

1. The above code assumes that the SPI status register (SPSR) can be left at its default values (SPI2X = 0). How would you explicitly clear this register in C++ and/or Assembly?
2. The above code does not include the instructions to initialize the Data Direction registers for DD_MOSI (Port B bit 3), the SPI clock DD_SCK (Port B bit 5), or our SS signal PB2 (Port B bit 2). How would you write the code in C++ and/or Assembly to initialize the SPI data direction register DDR_SPI (Port B DDR) so these pins were outputs?

How to Operate the SPI Subsystem – Polling –

The interconnection between Master and Slave consists of two shift Registers, and a Master clock generator. For our labs the ATmega’s SPI subsystem is the Master and the 74HC595 “8-Bit Shift Register with Output Latches” is the slave.



1. Writing a byte to the SPI Data Register (**SPDR**) starts the SPI clock generator, and the hardware shifts the eight bits into the Slave (74HC595). The Master generates the required clock pulses on the SCK line to interchange data.

   C Code Example

   ```
   /* Start Transmission */
   SPDR = LEDS;            // LEDS is an 8-bit variable in SRAM
   ```

   Assembly Code Example

   ```
   out  SPDR, spiLEDS      // spiLEDS is register r9
   ```

2. Data is always shifted from Master-to-Slave on the **M**aster **O**ut **S**lave **I**n (MOSI) line, and from Slave-to-Master on the **M**aster **I**n **S**lave **O**ut (MISO) line.

How to Operate the SPI Subsystem – Polling –

3. After shifting one byte, the SPI clock generator stops, setting the end of Transmission Flag (**SPIF** bit in the **SPSR** register). 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[3]. In our lab we used polling to monitor the status of the SPIF flag.

   C Code Example
   ```
   /* Wait for transmission complete */
   while(!(SPSR & (1<<SPIF)));
   ```

   Assembly Code Example
   ```
   wait: in    r16,SPSR
         bst   r16,SPIF
         brtc  wait
         ret
   ```

4. After the last data packet is transmitted, the Master will transfer the data to the eight-bit D-type storage register of the slave by strobing the slave select (SS) line. When configured as a Master, the SPI interface has no automatic control of the SS line. This must be handled by your software. Note: *We are using the SS line in a non-standard fashion. If you want to configure the ATmega328P to operate as a master, slave, or master/slave using the Atmel convention, please see the datasheet.*

   C Code Example
   ```
   /* Pulse SS line */
   PORTB |= (1 << (PB2));
   PORTB &= ~(1 << (PB2));
   ```

   Assembly Code Example
   ```
   sbi  PORTB,PB2
   cbi  PORTB,PB2
   ```

[3] ATmega328P Datasheet Figure 18-2 SPI Master-slave Interconnection

SPI Code Example

C++

```
void SPI_MasterInit(void)
{
  /* Set MOSI, SCK, and SS output, all others input */
  DDR_SPI = (1<<DD_MOSI)|(1<<DD_SCK)|(1<<DD_SS);

  /* Enable SPI, Master, set clock rate fck/16 */
  SPCR = (1<<SPE)|(1<<MSTR)|(1<<SPR0);
}

void SPI_MasterTransmit(char cData)
{
  /* SS Line Low */
  PORTB &= ~(1 << (PB2));

  /* Start transmission */
  SPDR = cData;

  /* Wait for transmission complete */
  while(!(SPSR & (1<<SPIF)));

  /* SS line High */
  PORTB |= (1 << (PB2));
}
```

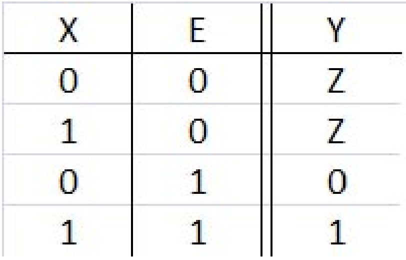
SPI Code Example – Assembly

```
; SPI interface registers
.DEF spiLEDS=r9
.DEF spi7SEG=r8

; Switches
.DEF switch=r7

InitShield:
   code to initialize protoshield GPIO ports is not included in this SPI code example
; Initialize SPI Port
   in    r16,DDRB       // Input from Port B Data Direction Register (DDRB) at i/o address 0x04
   sbr   r16,0b00101111 // Set PB5, PB3, PB2 (SCK, MOSI, SS) and PB1, PB0 (TEST LEDs) as outputs
   out   DDRB,r16       // Output to Port B Data Direction Register (DDRB) at i/o address 0x04
; Set SPCR Enable (SPE) bit 6, Master (MSTR) bit 4, clock rate fck/16 (SPR1 = 0,SPR0 = 1)
   ldi   r16,0b01010001
   out   SPCR,r16       // Output to SPI Control Register (SPCR) at i/o address 0x2c
   cbi   PORTB,2        // Clear I/O Port B bit 2 (SS) at i/o address 0x05
   ret

WriteDisplay:
   push  r16
; Start transmission of data (r16)
   cbi   PORTB,PB2    // ss line active low
   out   SPDR,spiLEDS
   rcall SpiTxWait
   out   SPDR,spi7SEG
   rcall spiTxWait
   sbi   PORTB,PB2    // ss line high
   pop   r16
   ret

SpiTxWait:
; Wait for transmission complete
   in    r16,SPSR
   bst   r16,SPIF
   brtc  spiTxWait
   ret
```

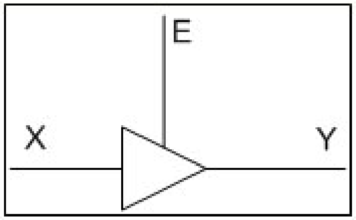
Appendix A Detail Description of the 74HC595

Let's look at the components that make up the 74HC595 shift register.[4]

Tri-State Output Buffers



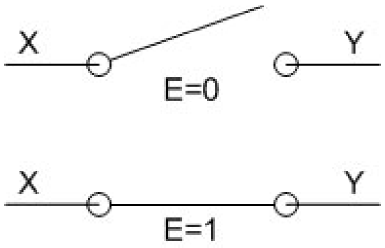
X	E	Y
0	0	Z
1	0	Z
0	1	0
1	1	1



The eight parallel-out pins of this shift register are driven by tri-state buffers. A tri-state buffer is a device commonly used on shift registers, memory, and many other kinds of integrated circuits.

The tri-state buffer shown above has two inputs, data (X) and control (E), which control the state of the output (Y). Just as the name implies, there are three output states: high, low and high impedance. When the pin labeled "E" is high, the output is equal to the input (Y=X).

Not very interesting right? Well, when the pin labeled "E" is low, the output is in high impedance mode. In high impedance mode, the output is virtually disconnected from the input, neither high nor low. The basic operation of a tri-state buffer can also be easily understood if compared to a switch. When the "E" pin is high, the switch is closed, and when the "E" pin is low, the switch is open. In the context of our shift register, the output pins will either contain our data or will be in high impedance mode.

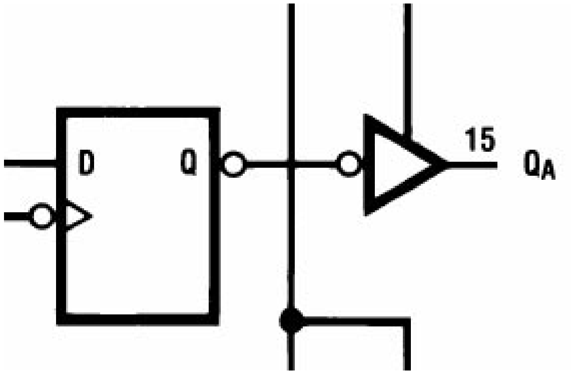


For more information regarding tri-state buffers, click here.
National Semiconductor - Tri-State Buffer IC

[4] The following Detail Description of the 74HC595 was written by Bryan Everett.

74HC595 Storage Registers (D-Flip Flops)

- Looking further into our shift register we see the storage registers. These registers are made up of D-type flip flops. The D-type flip flop is capable of storing one bit of memory. The D-flip flop's function is to place on the output whatever data is on it's input when the flip flop detects a rising edge signal (input buffer inverts clock before input of FF shown) on the clock port. works by placing the data to be stored (1 or 0) on the D pin. Once the data is on the D line, the clock pin must be pulsed high. On the rising edge of the pulse the data on the D pin will appear on the Q pin.



- In context to our shift register, when the data appears on D pins of the storage registers and is ready to be displayed, the clock pin is pulsed and the data is sent to the tri-state buffers.

For more information regarding D-type flip flops, click here.
Fairchild Semiconductors - D-Flip Flop

74HC595 Shift Registers (D-Flip Flops)

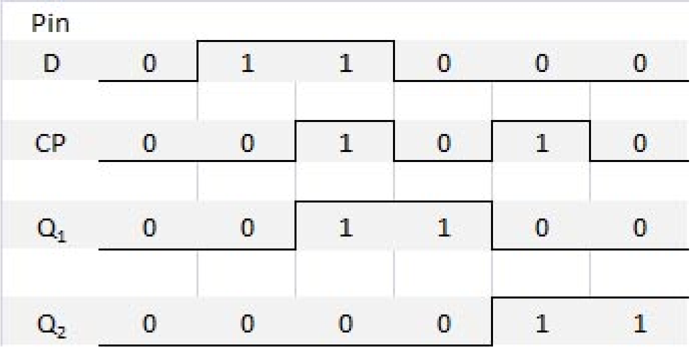
- The shift registers are final stage and are made up of D-Flip flops as well. These are the heart of our 74HC595 shift register. Here is a simplified version of what makes our shift registers work. What we have there is two D-type shift registers. The output of the first D flip flop is connected to the input of the second D flip flop. The clock pins are connected together on all D flip flops.

This image cannot currently be displayed.

- To understand how this shift register works, we will look at a two bit shift register:

- Suppose we want to set Q_2 high and Q_1 low:

1. The D pin is set high.
2. The clock pin is pulsed high once. (This makes the output Q_1 high. Q_1 is connected to the input of the second D flip flop)
3. The D pin is brought low.
4. The clock is pulsed once again.
5. The result is Q_1 = 0 and Q_2 = 1.



Pin						
D	0	1	1	0	0	0
CP	0	0	1	0	1	0
Q_1	0	0	1	1	0	0
Q_2	0	0	0	0	1	1

- The above example only covers a two bit shift register. See original logic diagram of our 74HC595 for an 8-bit shift register example.