AVR Microcontroller

Microprocessor Course

Chapter 17

SPI PROTOCOL and

MAX7221 DISPLAY INTERFACING

Day 1397

SECTION 17.1 : SPI BUS PROTOCOL

The SPI (Serial Peripheral Interface) is a bus interface connection incorporated into many devices such as ADC, DAC, and EEPROM.

In this section we examine the pins of SPI bus and show how the read and write operations in the SPI work.

The SPI bus was originally started by Motorola Corp. (now Freescale), but in recent years has become a widely used standard adapted by many semiconductor chip companies.

4 PINs of SPI (4 wire interface)

SPI devices use only 2 pins for data transfer, called **SDI** (Din) and **SDO** (Dout), instead of the 8 or more pins used in traditional buses.

The SPI bus has the **SCLK** (shift clock) pin to synchronize the data transfer between two chips.

The last pin of the SPI bus is **CE** (chip enable), which is used to initiate and terminate the data transfer.

These four pins, SDI, SDO, SCLK, and CE, make the SPI a 4-wire interface.

SECTION 17.1 : SPI BUS PROTOCOL

There is also a widely used standard called a 3-wire interface bus. In a 3-wire interface bus, we have SCLK and CE, and only a single pin for data transfer.

The SPI 4-wire bus can become a 3-wire interface when the SDI and SDO data pins are tied together.

However, there are some major differences between the SPI and 3-wire devices in the data transfer protocol. For that reason, a device must support the 3-wire proto-col internally in order to be used as a 3-wire device. Many devices such as the DS 1306 RTC (real-time clock) support both SPI and 3-wire protocols.

SPI PROTOCOL and MAX7221 DISPLAY INTERFACING
SECTION 17.1 : SPI BUS PROTOCOL

In many chips the SDI, SDO, SCLK, and CE signals are alternatively named as MOSI, MISO, SCK, and SS as shown in Figure 17-2 (compare with Figure 17-1).

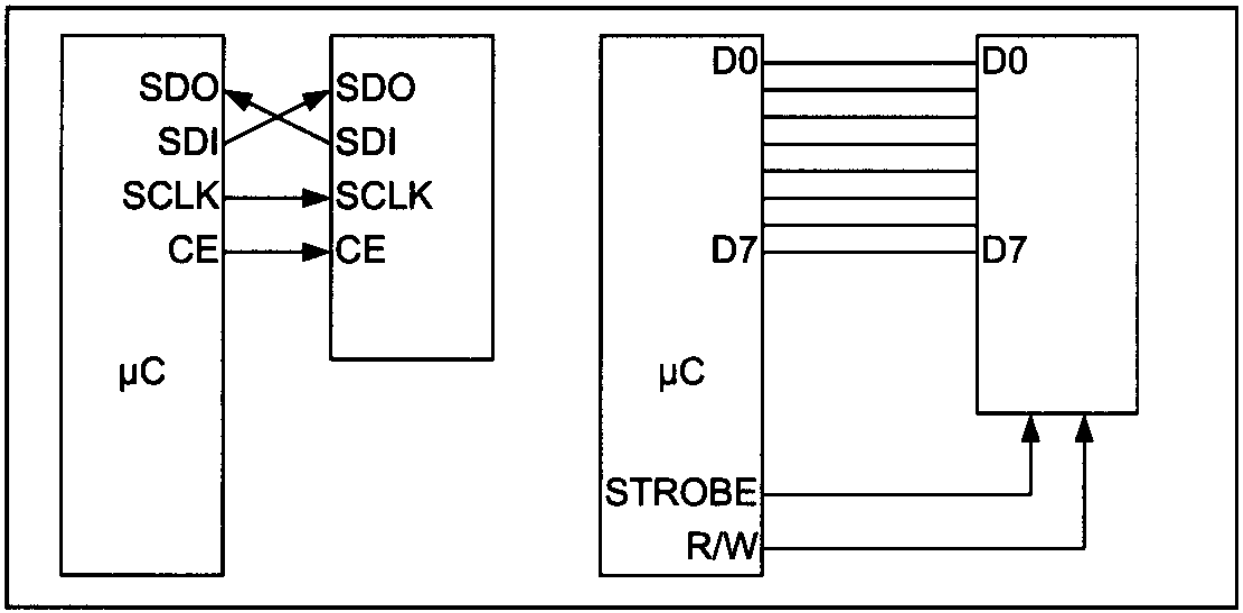


Figure 17-1. SPI Bus vs. Traditional Parallel Bus Connection to Microcontroller

SECTION 17.1 : SPI BUS PROTOCOL

How SPI works

SPI consists of two shift registers, one in the master and the other in the slave side.

Also, there is a clock generator in the master side that generates the clock for the shift registers.

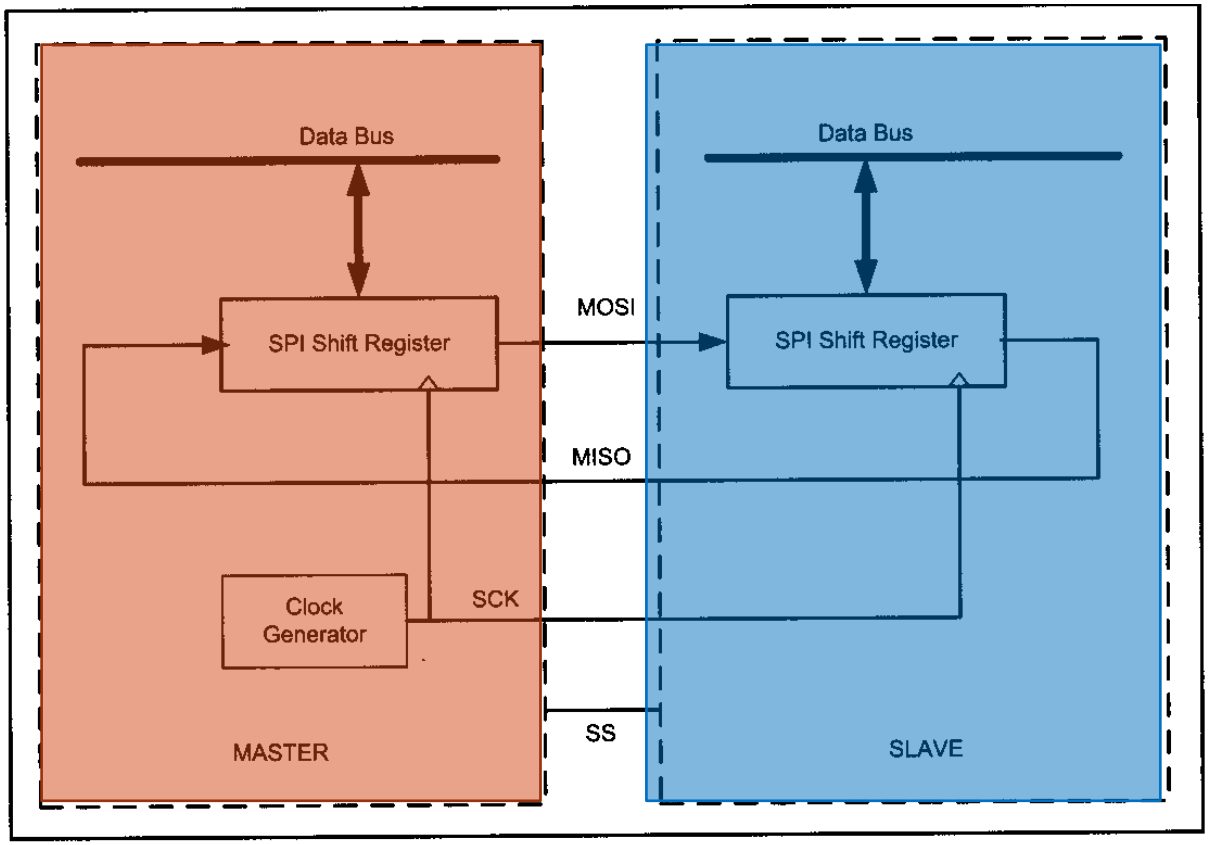


Figure 17-2. SPI Architecture

How SPI works

In SPI, the shift registers are 8 bits long. It means that after 8 clock pulses, the contents of the two shift registers are interchanged. When the master wants to send a byte of data, it places the byte in its shift register and generates 8 clock pulses. After 8 clock pulses the byte is transmitted to the other shift register. When the master wants to receive a byte of data, the slave side should place the byte in its shift register, and after 8 clock pulses the data will be received by the master shift register. It must be noted that SPI is full duplex, meaning that it sends and receives data at the same time.

SPI read and write

In connecting a device with an SPI bus to a microcontroller, we use the microcontroller as the master while the SPI device acts as a slave.

This means that the microcontroller generates the SCLK, which is fed to the SCLK pin of the SPI device.

SECTION 17.1 : SPI BUS PROTOCOL

The SPI protocol uses SCLK to synchronize the transfer of information one bit at a time, where the most-significant bit (MSB) goes in first. During the transfer, the CE must stay HIGH. The information (address and data) is transferred between the microcontroller and the SPI device in groups of 8 bits, where the address byte is followed immediately by the data byte.

To distinguish between the read and write operations, the D7 bit of the address byte is always 1 for write, while for the read, the D7 bit is LOW.

Clock polarity and phase in SPI device

In SPI communication, the master and slave(s) must agree on the clock polarity and phase with respect to the data. Freescale names these two options as CPOL (clock polarity) and CPHA (clock phase), respectively, and most companies like Atmel have adopted that conven-tion. At CPOL= 0 the base value of the clock is zero, while at CPOL = 1 the base value of the clock is one. CPHA = 0 means sample on the leading (first) clock edge, while CPHA = 1 means sample on the trailing (second) clock. Notice that if the base value of the clock is zero, the leading (first) clock edge, is the rising edge but if the base value of the clock is one, the leading (first) clock edge is falling edge

SECTION 17.1 : SPI BUS PROTOCOL

CPOL	CPHA	Data Read and Change Time	SPI Mode
0	0	Read on rising edge, changed on a falling edge	0
0	1	Read on falling edge, changed on a rising edge	1
1	0	Read on falling edge, changed on a rising edge	2
1	1	Read on rising edge, changed on a falling edge	3

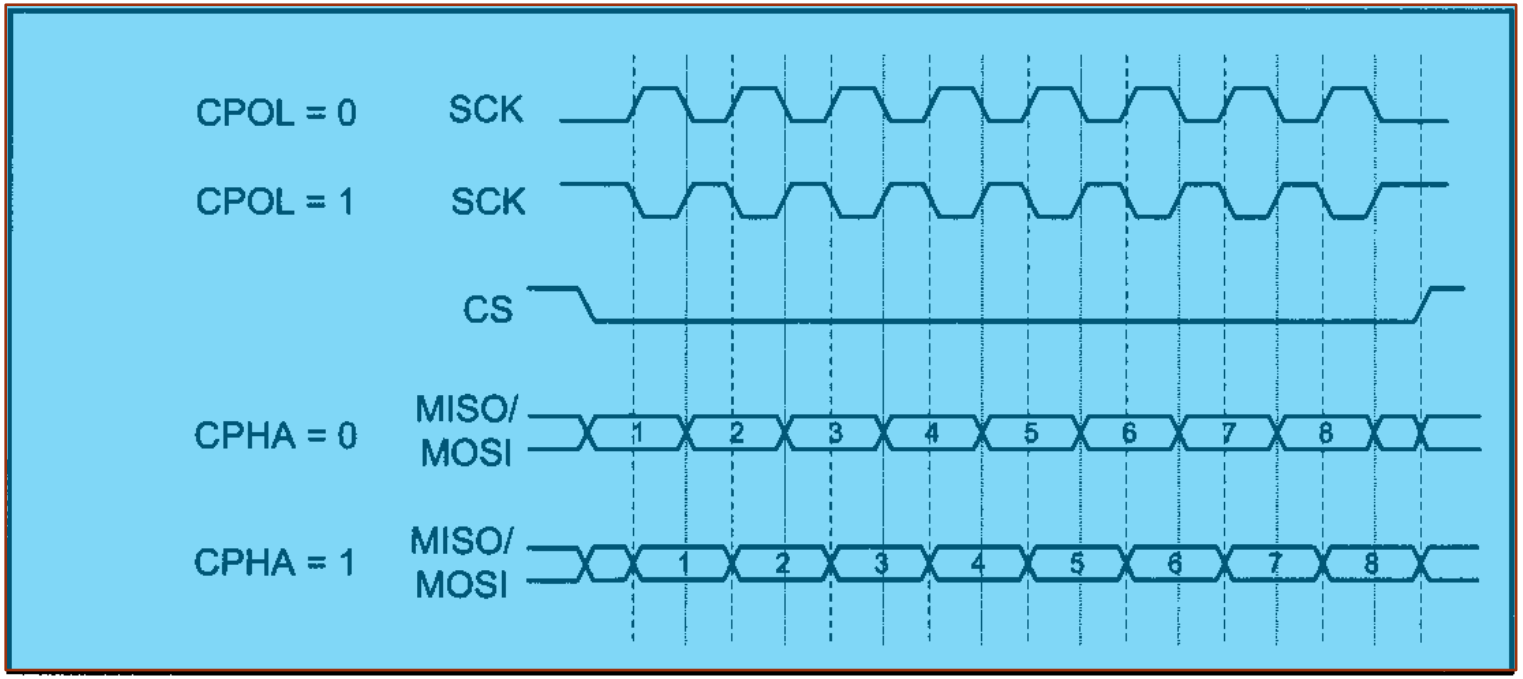


Figure 17-3. SPI Clock Polarity and Phase

SECTION 17.1 : SPI BUS PROTOCOL

Steps for writing data to an SPI device

In accessing SPI devices, we have two modes of operation: single-byte and multibyte.

Single-byte write

The following steps are used to send (write) data in single-byte mode for SPI devices, as shown in Figure 17-4:

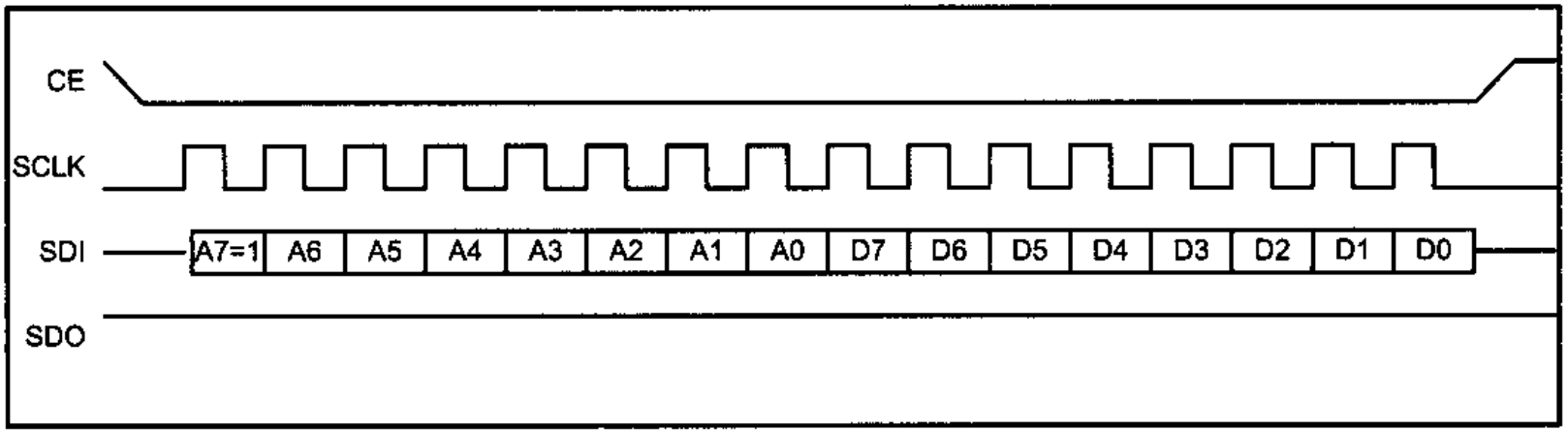


Figure 17-4. SPI Single-Byte Write Timing (Notice A7 = 1)

SECTION 17.1 : SPI BUS PROTOCOL

1. Make CE = 0 to begin writing.
2. The 8-bit address of the first location is provided and shifted in, one bit at a time, with each edge of SCLK. Notice that A7 = 1 for the write operation and the A7 bit goes in first.
3. The 8-bit data for the first location is provided and shifted in, one bit at a time, with each edge of the SCLK. From then on, we simply provide consecutive bytes of data to be placed in consecutive memory locations. In the process, CE must stay low to indicate that this is a burst mode multibyte write operation.
4. Make CE = 1 to end writing.

Multibyte burst write

Burst mode writing is an effective means of loading consecutive locations. In burst mode, we provide the address of the first location, followed by the data for that location. From then on, while CE = 0, consecutive bytes are written to consecutive memory locations.

In this mode, the SPI device internally increments the address location as long as CE is LOW. The following steps are used to send (write) multiple bytes of data in burst mode for SPI devices as shown in Figure 17-5:

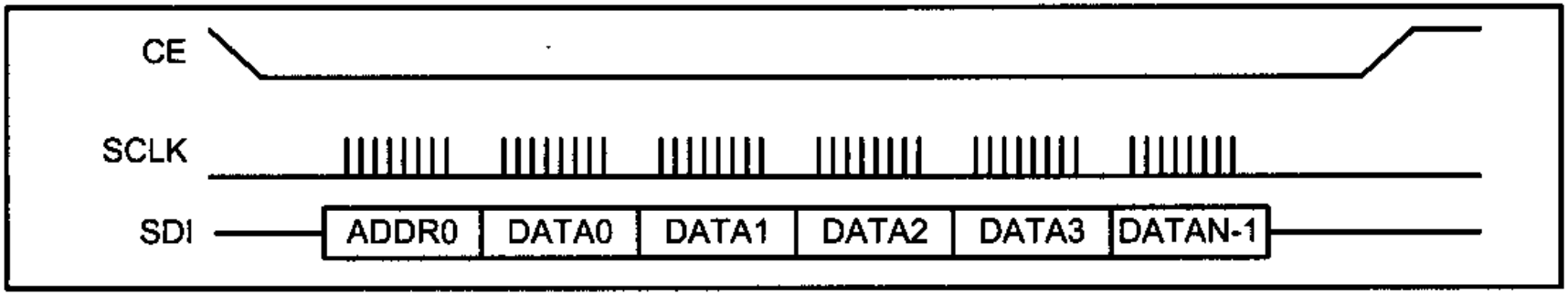


Figure 17-5. SPI Burst (Multibyte) Mode Writing

SECTION 17.1 : SPI BUS PROTOCOL

1. Make CE = 0 to begin writing.
2. The 8-bit address is shifted in, one bit at a time, with each edge of SCLK. Notice that A7 = I for the write operation, and the A7 bit goes in first.
3. After all 8 bits of the address are sent in, the SPI device expects to receive the data belonging to that address location immediately.
4. The 8-bit data is shifted in one bit at a time, with each edge of the SCLK.
5. Make CE = 1 to indicate the end of the write cycle.

SECTION 17.1 : SPI BUS PROTOCOL

Steps for reading data from an SPI device

In reading SPI devices, we also have two modes of operation:

single-byte and multibyte.

Single-byte read The following steps are used to get (read) data in single-byte mode from SPI devices, as shown in Figure 17-6:

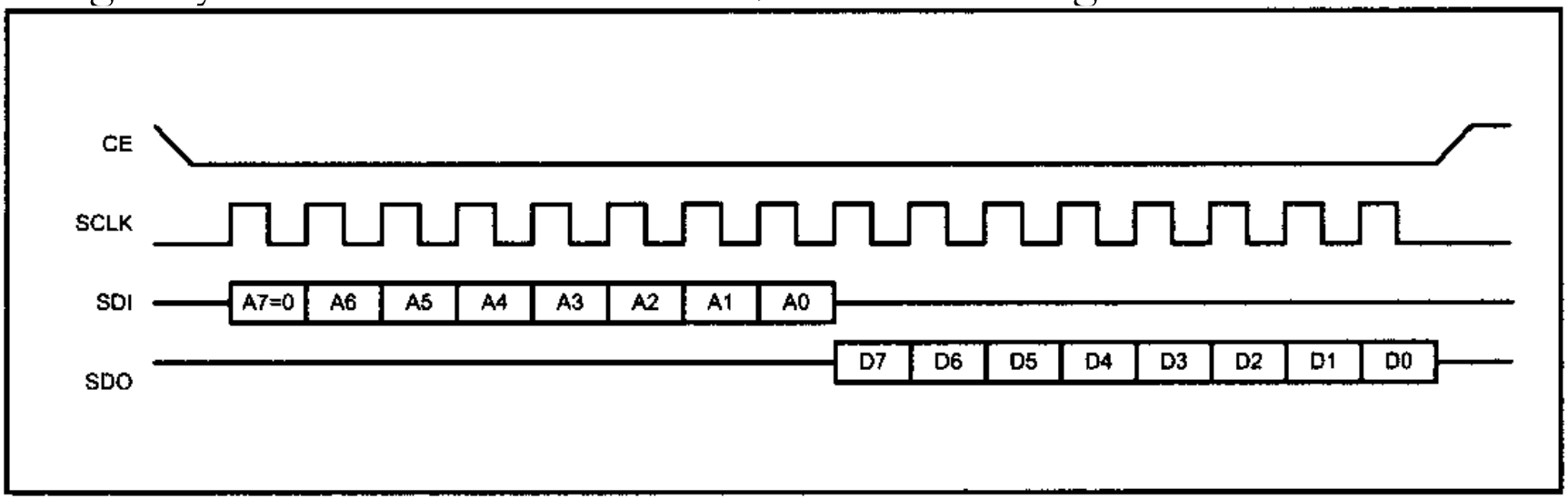


Figure 17-6. SPI Single-Byte Read Timing (Notice A7 = 0)

SECTION 17.1 : SPI BUS PROTOCOL

1. Make CE = 0 to begin reading.
2. The 8-bit address is shifted in one bit at a time, with each edge of SCLK. Notice that A7 = 0 for the read operation, and the A7 bit goes in first.
3. After all 8 bits of the address are sent in, the SPI device sends out data belonging to that location.
4. The 8-bit data is shifted out one bit at a time, with each edge of the SCLK.
5. Make CE = 1 to indicate the end of the read cycle.

Multibyte burst read

Burst mode reading is an effective means of bringing out the contents of consecutive locations.

In burst mode, we provide the address of the first location only. From then on, while CE = 0, consecutive bytes are brought out from consecutive memory locations.

In this mode, the SPI device internally increments the address location as long as CE is LOW. The following steps are used to get (read) multiple bytes of data in burst mode for SPI devices, as shown in Figure 17-7:

SECTION 17.1 : SPI BUS PROTOCOL

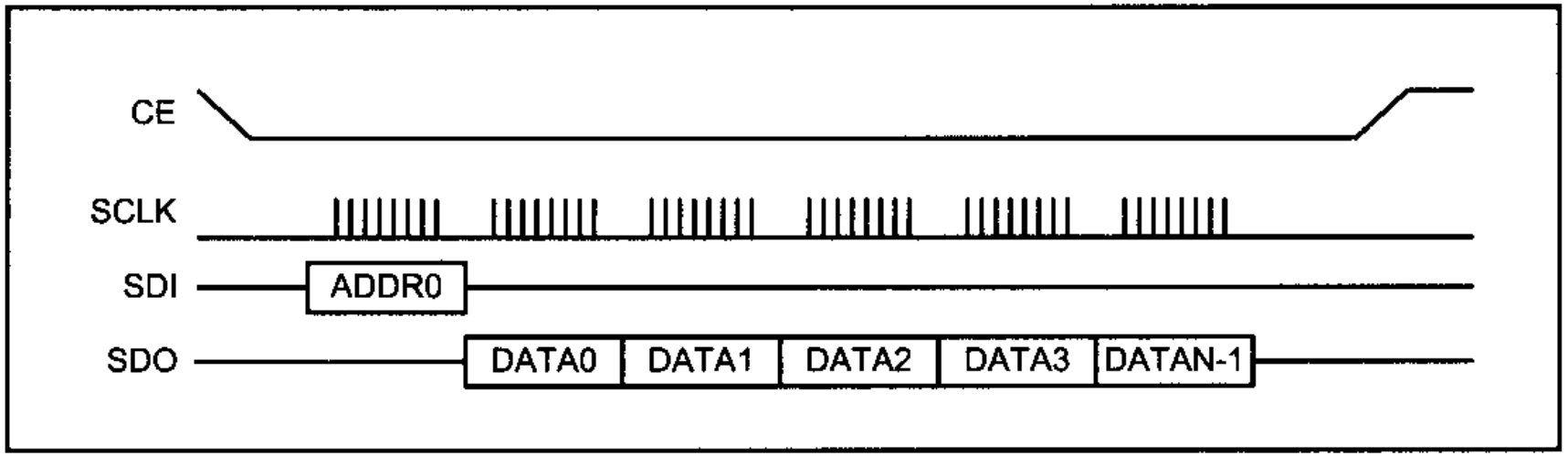


Figure 17-7. SPI Burst (Multibyte) Mode Reading

Multibyte burst read

Burst mode reading is an effective means of bringing out the contents of consecutive locations.

In burst mode, we provide the address of the first location only. From then on, while CE = 0, consecutive bytes are brought out from consecutive memory locations.

In this mode, the SPI device internally increments the address location as long as CE is LOW. The following steps are used to get (read) multiple bytes of data in burst mode for SPI devices, as shown in Figure 17-7:

SECTION 17.1 : SPI BUS PROTOCOL

1. Make CE = 0 to begin reading.
2. The 8-bit address of the first location is provided and shifted in, one bit at a time, with each edge of SCLK. Notice that A7 = 0 for the read operation, and the A7 bit goes in first.
3. The 8-bit data for the first location is shifted out, one bit at a time, with each edge of the SCLK. From then on, we simply keep getting consecutive bytes of data belonging to consecutive memory locations. In the process, CE must stay LOW to indicate that this is a burst mode multibyte read operation.
4. Make CE = 1 to end reading.

SECTION 17.2 : SPI PROGRAMMING INB AVR

Most AVRs, including ATmega family members, support SPI protocols. In AVR three registers are associated with SPI. They are SPSR (SPI Status Register), SPCR (SPI Control Register), and SPDR (SPI Data Register).

SPI Control Register – SPCR

Bit	7	6	5	4	3	2	1	0	
	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 Status Register – SPSR

Bit	7	6	5	4	3	2	1	0	
	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 Data Register – SPDR

Bit	7	6	5	4	3	2	1	0	
	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	Undefine

SECTION 17.2 : SPI PROGRAMMING INB AVR

Figure 17-8 shows the bits of the SPSR register used for SPI.

SPIF	WCOL	-	-	-	-	-	SPI2X

Bit 7 – SPIF (SPI Interrupt Flag)
In master mode, this bit is set in two situations: when a serial transfer is completed, or when SS pin is an input and is driven low by an external device. Setting the SPIF flag to one will cause an interrupt if SPIE in SPCR is set and global interrupts are enabled.

Bit 6 – WCOL (Write COLlision Flag)
The WCOL bit is set if you write on SPDR during a data transfer.

Bit 0 – SPI2X (Double SPI Speed)
When the SPI is in master mode, setting this bit to one doubles the SPI speed.

Notice that both the WCOL bit and the SPIF bit are cleared when you read the SPI Status Register and then access the SPI Data Register. Alternatively, the SPIF bit is cleared by hardware when executing the corresponding interrupt handler.

Figure 17-8. SPI Status Register *(Note: The portion shown is used for SPI.)*

SECTION 17.2 : SPI PROGRAMMING INB AVR

SPIE	SPE	DORD	MSTR	CPOL	CPHA	SPR1	SPR0

Bit 7 – SPIE: SPI Interrupt Enable
Setting this bit to one enables the SPI interrupt.

Bit 6 – SPE: SPI Enable
Setting this bit to one enables the SPI.

Bit 5 – DORD: Data Order
This bit lets you choose to either transmit MSB and then LSB or vice versa. The LSB is transmitted first if DORD is one; otherwise, the MSB is transmitted first.

Bit 4 – MSTR: Master/Slave Select
If you want to work in master mode then set this bit to one; otherwise, slave mode is selected. Notice that if the SS pin is configured as an input and is driven low while MSTR is set, MSTR will be cleared, and SPIF will become set.

SECTION 17.2 : SPI PROGRAMMING INB AVR

Bit 3 – CPOL: Clock Polarity
This bit set the base value of clock when it is idle. At CPOL = 0 the base value of the clock is zero while at CPOL = 1 the base value of the clock is one.

Bit 2 – CPHA: Clock Phase
CPHA = 0 means sample on the leading (first) clock edge, while CPHA = 1 means sample on the trailing (second) clock.

Bits 1, 0 – SPR1, SPR0: SPI Clock Rate Select 1 and 0
These two bits control the SCK rate of the device in master mode. See Table 17-2.

Figure 17-9. SPI Control Register

SECTION 17.2 : SPI PROGRAMMING INB AVR

In Table 17-2 you see how SPI2X, SPR1, and SPRO are combined to make different clock frequencies for master. As you see in Table 17-2, by setting SPI2X to one, the SCK frequency is doubled.

Table 17-2: SCK Frequency

SPI2X	SPR1	SPR0	SCK Frequency
0	0	0	Fosc/4
0	0	1	Fosc/16
0	1	0	Fosc/64
0	1	1	Fosc/128
1	0	0	Fosc/2 (Not recommended!)
1	0	1	Fosc/8
1	1	0	Fosc/32
1	1	1	Fosc/64

SPDR (The SPI Data Register)

The SPI Data Register is a read/write register. To write into SPI shift register, data must be written to SPDR.

To read from the SPI shift register, you should read from SPDR. Writing to the SPDR register initiates data transmission. Notice that you cannot write to SPDR before the last byte is transmitted completely, otherwise a collision will happen. You can read the received data before another byte of data is received completely.

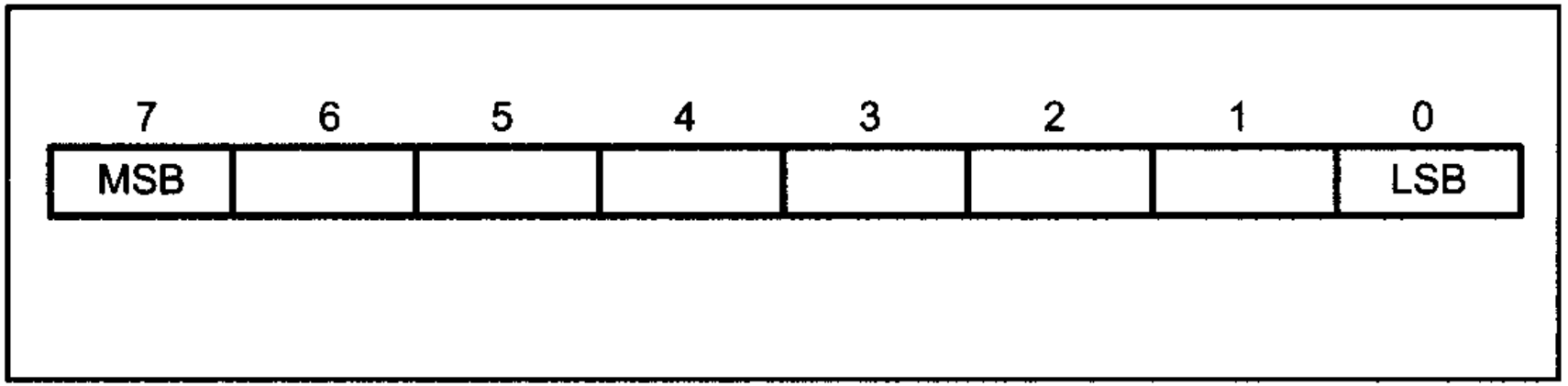


Figure 17-10. SPI Data Register

SS pin in AVR

As we mentioned before, the Slave Select (SS) pin of the SPI bus is used to initiate and terminate the data transfer.

When you are in master mode, you can choose to make this pin either input or output. If you make it output, the SPI circuit of AVR will not control the SS pin and you can make it one or zero by software.

When you make the SS pin an input, it will control the function of SPI. In this case you should externally make SS pin high to ensure master SPI operation. If an external device makes the SS pin low, the SPI module stops working in master mode and switches to slave mode by clearing the MSTR bit in SPCR, and then sets the SPIF bit in SPSR. It is highly recommended to make the SS pin output if you do not want to be interrupted when you are working in master mode.

SECTION 17.2 : SPI PROGRAMMING INB AVR

When you are in slave mode, the SS pin is always input and you cannot control it by software. You should hold it externally low to activate the SPI. When SS is driven high, SPI is disabled and all pins of SPI are input.

Also the SPI module will immediately clear any partially received data in the shift register. As we mentioned before, it can be used in packet synchronizing by initiating and terminating the data transfer. Notice that when you are working in slave mode and the SS pin is driven high by an external device, the SPI module is reset but not disabled and it is not necessary to enable it again.

SECTION 17.2 : SPI PROGRAMMING INB AVR

SPI programming in AVR

Before you start data transmission, you should set SPI Mode (Clock Polarity and Clock Phase) by setting the values of the CPOL and CPHA bits in SPCR. See Table 17-1.

In AVR you can operate in either master or slave modes.

Master operating mode

If you want to work in master mode, you should set the MSTR bit to one. Also you should set SCK frequency by setting the values of SPI2X, SPR1, and SPR2 according to Table 17-2.

Then you should enable SPI by setting the SPIE bit to one before you start data transmission. Writing a byte to the SPI Data Register (SPDR) starts data exchange by starting the SPI clock generator. After shifting the last (8th) bit, the SPI clock generator stops and the SPIF flag changes to one.

SECTION 17.2 : SPI PROGRAMMING INB AVR

The byte in the master shift register and the byte in the slave shift register are exchanged after the last clock. Notice that you cannot write to the SPI Data Register before transmission is completed, otherwise the collision happens.

To get the received data you should read it from SPDR before the next byte arrives. We can use interrupts or poll the SPIF to know when a byte is exchanged. See Example 17-1.

As we mentioned before, in case of multibyte burst write, the master continues to shift the next byte by writing it into SPDR. If you want to signal the end of the packet, you should pull high the SS pin.

When AVR is configured as a master, the SPI will not control the SS pin. If you want to make SS high or low, you have to do it by writing 1 or 0, respectively, to the SS bit of Port B.

SECTION 17.2 : SPI PROGRAMMING INB AVR

Example 17-1

Write an AVR program to initialize the SPI for master, mode 0, with CLCK frequency = Fosc/16, and then transmit 'G' via SPI repeatedly. The received data should be displayed on Port A.

Solution:

```
.INCLUDE "M32DEF.INC"
.equ  MOSI = 5                          ;for ATmega32
.equ  SCK = 7
.equ  SS = 4

      LDI   R17,0xFF                    ;Port A is output
      OUT   DDRA,R17

      LDI   R17,(1<<MOSI)|(1<<SCK)|(1<<SS)
      OUT   DDRB,R17                    ;MOSI,SCK, and SS output
```

SPI PROTOCOL and MAX7221 DISPLAY INTERFACING
SECTION 17.2 : SPI PROGRAMMING INB AVR

```
      LDI   R17,(1<<SPE)|(1<<MSTR)|(1<<SPR0) ;enable SPI
      OUT   SPCR,R17                ;master, CLK = fck/16
Transmit:
      CBI   PORTB,SS                ;enable slave device
      LDI   R17,'G'                 ;move G letter to R17
      OUT   SPDR,R17                ;start transmission of G
Wait:
      SBIS  SPSR,SPIF               ;wait for transmission
      RJMP  Wait                    ;to complete
      IN    R18,SPDR                ;read received data into R18
      OUT   PORTA,R18               ;move R18 to PORTA

      SBI   PORTB,SS                ;disable slave device
      RJMP Transmit                 ;do it again
```

Slave operating mode

When AVR is configured as a slave, the function of the SPI interface depends on the SS pin. If the SS is driven high, MISO is tri-stated and the SPI interface sleeps. Only the contents of SPDR may be updated in this state.

When SS is driven low, the data will be shifted by incoming clock pulses on the SCK pin. SPIF changes to one when the last bit of a byte has been shifted completely. Notice that the slave can place new data to be sent into SPDR before reading the incoming data; this is because in AVR there are two one-byte buffers to store received data.

SECTION 17.2 : SPI PROGRAMMING INB AVR

In slave mode there is no need to set SCK frequency because the SCK is generated by the master, but you must select the SPI mode (Clock Phase and Clock Polarity) and Data Order to match with SPI mode and Data Order of the other side (master device). Finally you should enable the SPI by setting the SPIE bit of SPCR to one. See Example 17-2. Notice that Example 17-2 is the slave version of Example 17-1.

Example 17-2

Write an AVR program to initialize the SPI for slave, mode 0, with CLCK frequency = fck/16, and then transmit 'G' via SPI repeatedly. The received data should be displayed on Port A.

Solution:

```
.INCLUDE "M32DEF.INC"
.equ  MISO = 6
```

SPI PROTOCOL and MAX7221 DISPLAY INTERFACING
SECTION 17.2 : SPI PROGRAMMING INB AVR

```
      LDI   R17,0xFF              ;Port A is output
      OUT   DDRA,R17
      LDI   R17,(1<<MISO)         ;MISO is output
      OUT   DDRB,R17

      LDI   R17,(1<<SPE)          ;enable SPI slave mode 0
      OUT   SPCR,R17              ;
Again:
      LDI   R17,'G'               ;move letter G to R17
      OUT   SPDR,R17              ;send data to SPDR to be
                                  ;transmitted
Wait:
      SBIS  SPSR,SPIF             ;skip next instruction if IF=1
      RJMP  Wait                  ;otherwise jump wait
      IN    R18,SPDR              ;read received data into R18
      OUT   PORTA,R18             ;send R18 to PORTA
      RJMP  Again                 ;do it again
;
;It must be noted that slave will not start transfer or
;receive until it senses the clock from master
```

SPI programming in C for AVR

Example 17-3 (C version of 17-1)

Rewrite Example 17-1 in C.

Solution:

```
#include <avr/io.h>                          //standard AVR header
#define MOSI 5
#define SCK 7
int main (void)
{
   DDRB = (1<<MOSI)|(1<<SCK);                //MOSI and SCK are output
   DDRA = 0xFF;                              //Port A is output
   SPCR = (1<<SPE)|(1<<MSTR)|(1<<SPR0);//enable SPI as master
   while(1){                                 //do for ever
      SPDR = 'G';                            //start transmission
      while(!(SPSR & (1<<SPIF)));            //wait transfer finish
      PORTA = SPDR;                          //move received data to
   }                                         //Port A
   return 0;
}
```

SECTION 17.2 : SPI PROGRAMMING INB AVR

Example 17-4 (C version of 17-2)

Rewrite Example 17-2 in C.

Solution:

```
#include <avr/io.h>                    //standard AVR header
#define MISO 6
int main (void)
{
   DDRA = 0xFF ;                       //Port A is output
   DDRB = (1<<MISO);                   //MISO is output
   SPCR = (1<<SPE);                    //enable SPI as slave
   while(1){
      SPDR = 'G';
      while(!(SPSR &(1<<SPIF)));       //wait for transfer finish
      PORTA = SPDR;                    //move received data to PORTA
   }
   return 0;
}
```

What is a 7-segment display?

In many applications, when you want to display numbers, 7-segments are the best choice. These displays are made of 7 LEDs to show different numbers plus another LED to display the decimal point. Some characters like A, b, c, d, E, and Ti are also displayed by 7-segments. Figure 17-12 shows how to display digits.

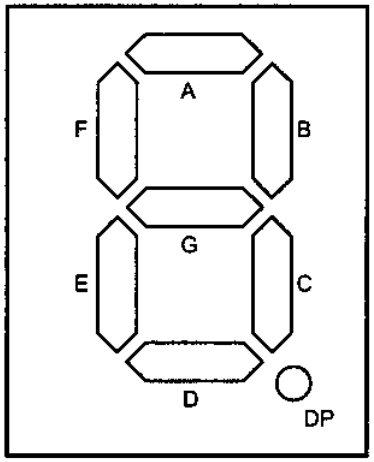


Figure 17-11. 7-segment

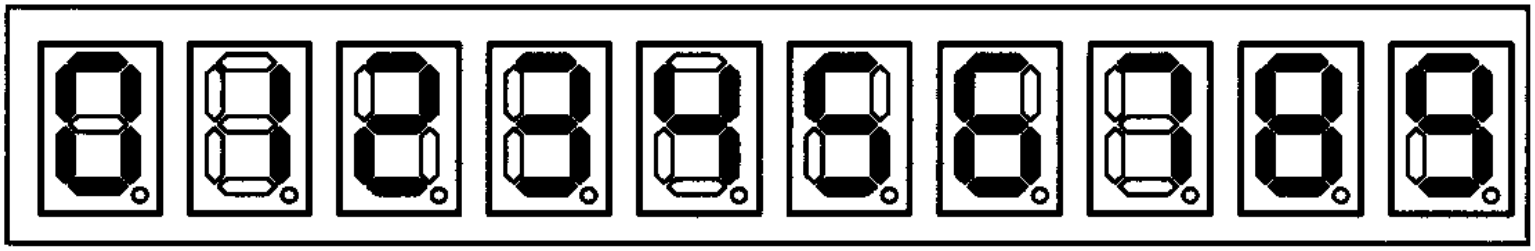


Figure 17-12. 7-Segment Display

There are two types of 7-segments, common anode and common cathode.

SECTION 17.3 : MAX7221 INTERFACING AND PROGRAMMING

MAX7221

In many applications you need to connect two or more 7-segment LEDs to a microcontroller. For example, if you want to connect four 7-segment LEDs directly to a microcontroller you need 4 x 8 = 32 pins. This is not feasible. The MAX7221 IC is an ideal chip for such applications since it supports up to eight 7-segment LEDs. We can connect the MAX7221 to the AVR chip using SPI proto-col and control up to eight 7-segment LEDs.

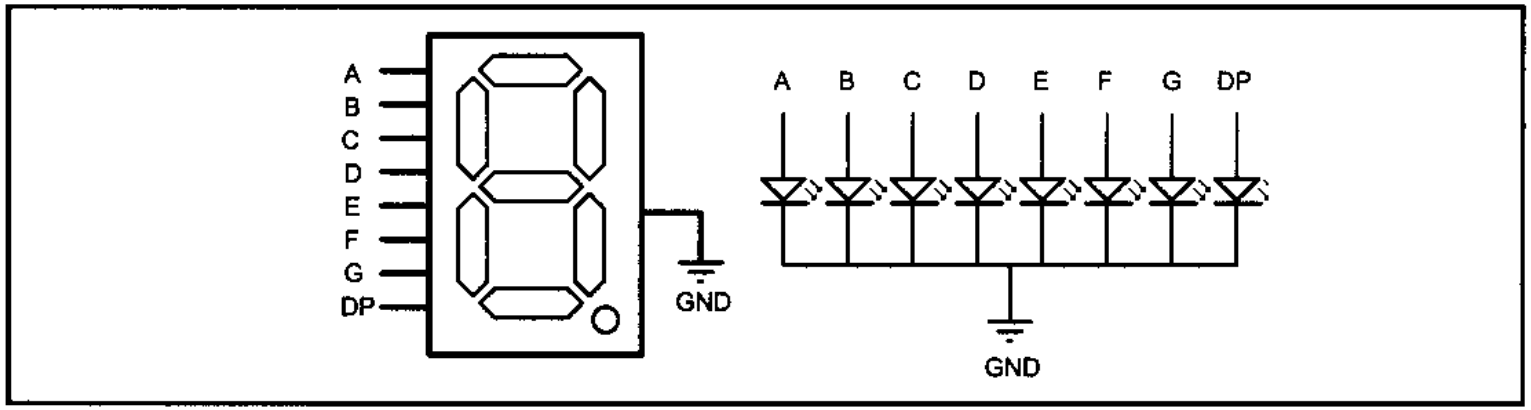


Figure 17-13. Common Cathode Connections in a 7-Segment Display

SECTION 17.3 : MAX7221 INTERFACING AND PROGRAMMING

The MAX7221 contains an internal decoder that can be used to convert binary numbers to 7-segment codes. That means we do not need to refresh the 7-segment LEDs. All you need to do is to send a binary number to the MAX7221, and the chip decodes the binary data and dis-plays the number. The device includes analog and digital brightness control, an 8x 8 static RAM that stores each digit, and a test mode that forces all LEDs on.

SECTION 17.3 : MAX7221 INTERFACING AND PROGRAMMING

MAX7221 pins and connections

The MAX7221 is a 24-pin DIP chip. It can be directly connected to the AVR and control up to eight 7-segment LEDs. A resistor or a potentiometer is the only external component that you need.

GND

Pin 4 and pin 9 are the ground. Notice that both of the ground pins should be connected to system ground and you can-not leave any of them unconnected.

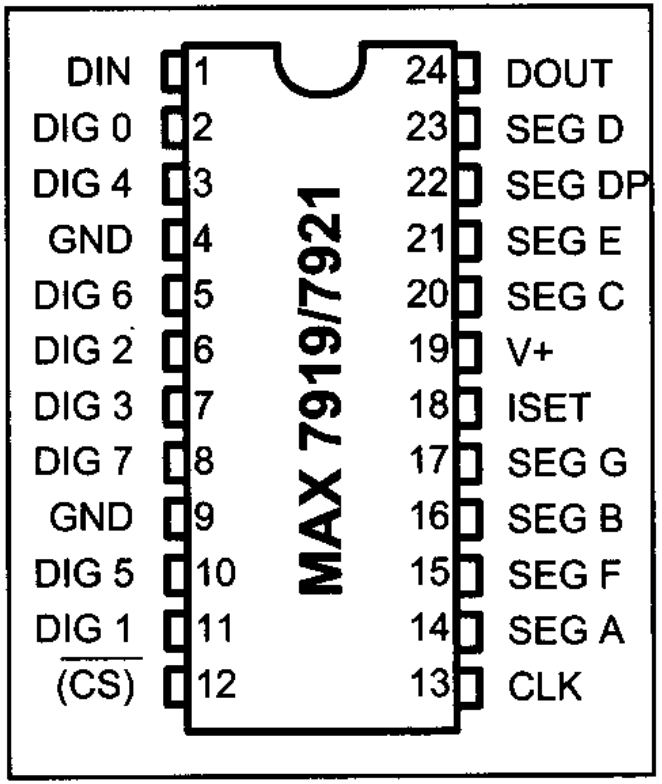


Figure 17-14. MAX7221

VCC

Pin 19 is the VCC and should be connected to the +5 V power supply. Notice that this pin is also the power to drive the 7-segments and the connecting wire to this pin should be able to handle 100-300 mA.

ISET

ISET Pin 18 is ISET and sets the maximum segment current. This pin should be connected to VCC through a resistor. A 10 kilo ohm resistor can be connected to this pin. If you want to manually control the intensity of the segments' light, you can replace the resistor with a 50K potentiometer. For more details about how to calculate the value of the resistor you can look at the datasheet of the chip.

CS

Pin 12 is the chip select pin and should be connected to the SS pin of the AVR. Serial data is loaded into the chip while CS is low, and the last 16 bits of the serial data are latched on CS's rising edge.

DIN

Pin 1 is the serial data input and should be connected to the MOSI pin of the AVR. On CLK's rising edge, data on this pin is loaded into the internal shift register. Notice that the MAX7221 uses the SPI Mode 0, that is, read on rising edge and change on falling edge as shown in Table 17-1.

CLK

Pin 13 is the serial clock input and should be connected to the SCK pin of the AVR. On MAX7221 the clock input is inactive when CS is high.

DOUT

Pin 24 is the serial data output and is used to connect more than one MAX7221 to a single SPI bus.

DIGO-DIG7

The DIG pins are the 7-segment selector pins and should be connected to the 7-segments' common cathode pin. The MAX7221 chip can control up to eight 7-segment LEDs. These eight 7-segment displays are designated as DIGO to DIG7.

SEGA--SEGG and DP

These pins select each segment and should be connected to segments of each 7-segment accordingly. Figure 17-15 shows the connection for two 7-seg-ments. You can connect up to eight 7-segments to MAX7221.

SECTION 17.3 : MAX7221 INTERFACING AND PROGRAMMING

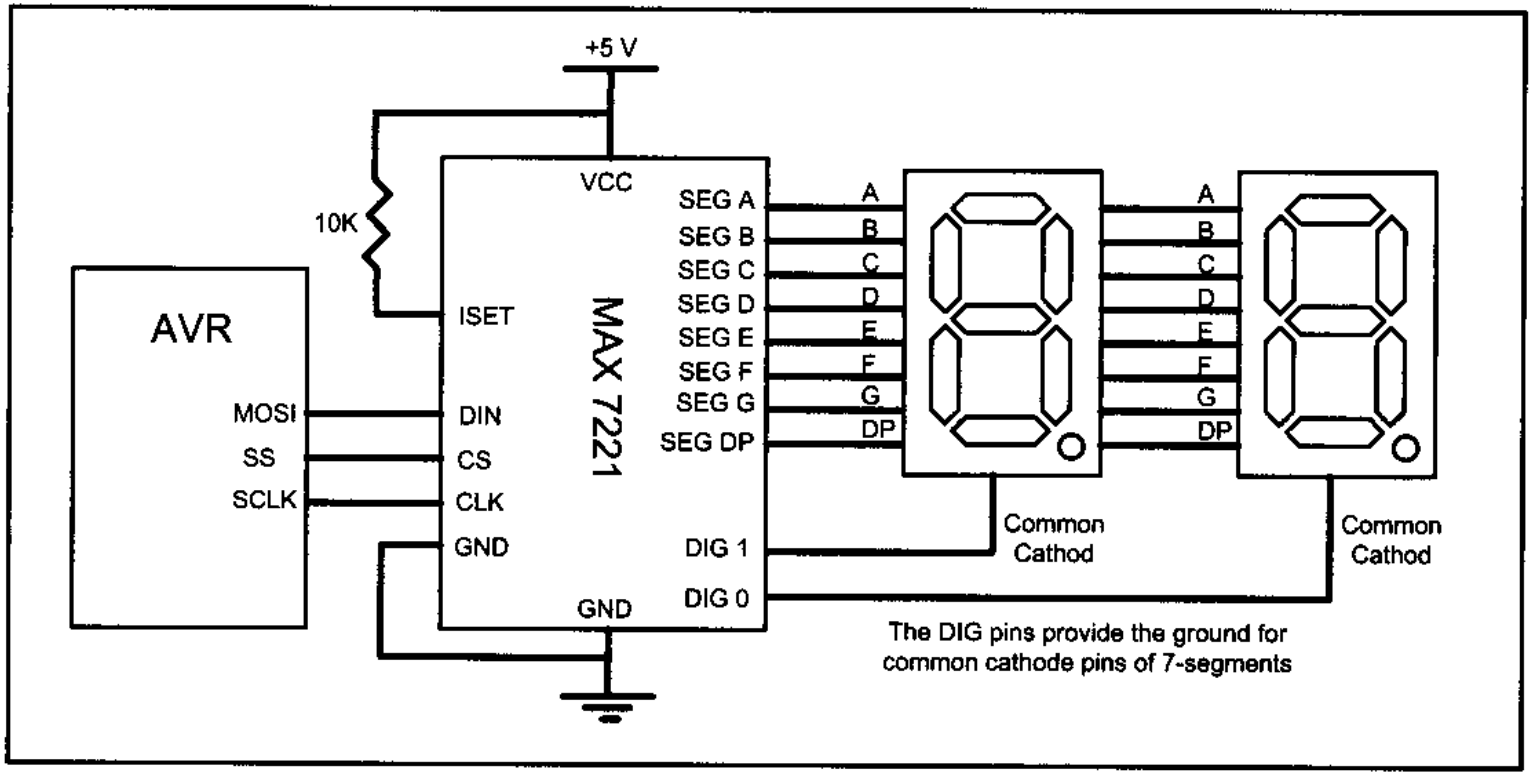


Figure 17-15. MAX7221 Connections to the AVR

MAX7221 data packet format

In MAX7221, data packets are 16 bits long (two bytes). You should first make CS low before transmitting; then you transmit two bytes of data and terminate the transmission by making CS high.

The first byte (MSBs) of each packet contains the command control bits, and the second byte is the data to be displayed. See Figure 17-16. The upper four bits (D15-D12) of the command byte are don't care and the lower four bits (D11-D8) are used to identify the meaning of the data byte to be followed. The second byte (D7—DO) of the two-byte packet is called the data byte and is the actual data to be displayed or control the 7-segment driver. Table 17-3 shows the binary and hex values of each command.

SECTION 17.3 : MAX7221 INTERFACING AND PROGRAMMING

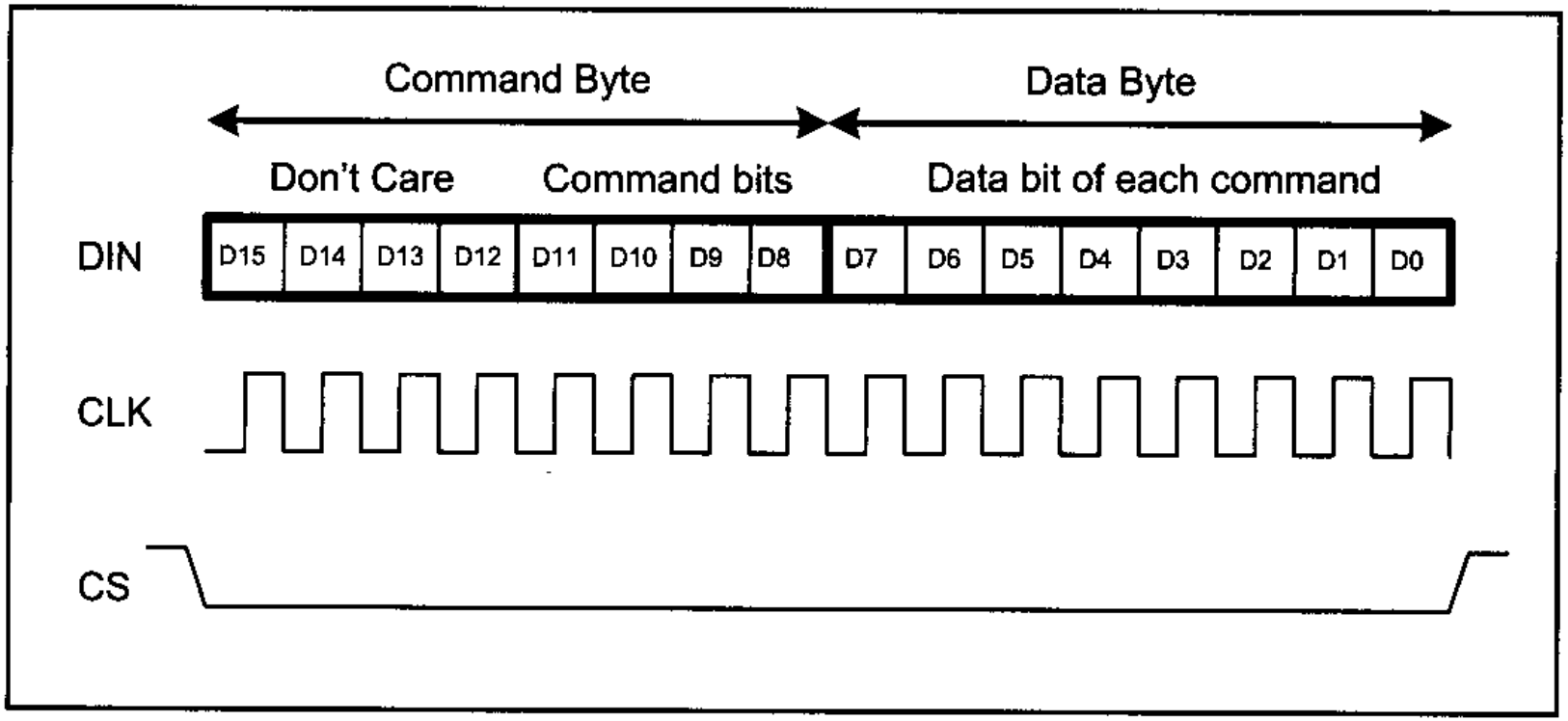


Figure 17-16. MAX7221 Packet Format

SECTION 17.3 : MAX7221 INTERFACING AND PROGRAMMING

Table 17-3: List of Commands in MAX7221

Command	D15-12	D11	D10	D9	D8	Hex Code
No operation	X	0	0	0	0	X0
Set value of digit 0	X	0	0	0	1	X1
Set value of digit 1	X	0	0	1	0	X2
Set value of digit 2	X	0	0	1	1	X3
Set value of digit 3	X	0	1	0	0	X4
Set value of digit 4	X	0	1	0	1	X5
Set value of digit 5	X	0	1	1	0	X6
Set value of digit 6	X	0	1	1	1	X7
Set value of digit 7	X	1	0	0	0	X8
Set decoding mode	X	1	0	0	1	X9
Set intensity of light	X	1	0	1	0	XA
Set scan limit	X	1	0	1	1	XB
Turn on/ off	X	1	1	0	0	XC
Display test	X	1	1	1	1	XF

Notes: 1) X means do not care.

2) Digits are designated as 0–7 to drive total of eight 7-segment LEDs.

Set value of digit 0-digit 7 (commands X1-X8)

These commands set what is to be displayed on each 7-segment. You can either send a binary number to the chip decoder and let it turn on/off the segments accordingly, or you may decide to turn on/off each segment of the 7-segment by yourself

The first way is useful when you do not want to deal with converting a binary number to 7-segment codes.

The second way is useful when you want to show a character or any other thing that is not predefined. For example, if you want to show U, you should use the second way and turn on/off segments your-self. Next, you will see how to enable or bypass the decoder for each 7-segment.

Set decoding mode (command X9)

This command lets you enable or bypass the binary to 7-segment decoding function for each 7-segment. Each bit in the data byte (second byte) is assigned to one digit (7-segment). D0 is assigned to Digit 0, D1 is assigned to Digit 1, and so on.

If you want to enable the decoding function for a digit you should set to one the bit assigned to that digit, and if you want to disable the decoding function you should clear the bit for that digit. Figure 17-17 shows the structure of the set decoding mode command.

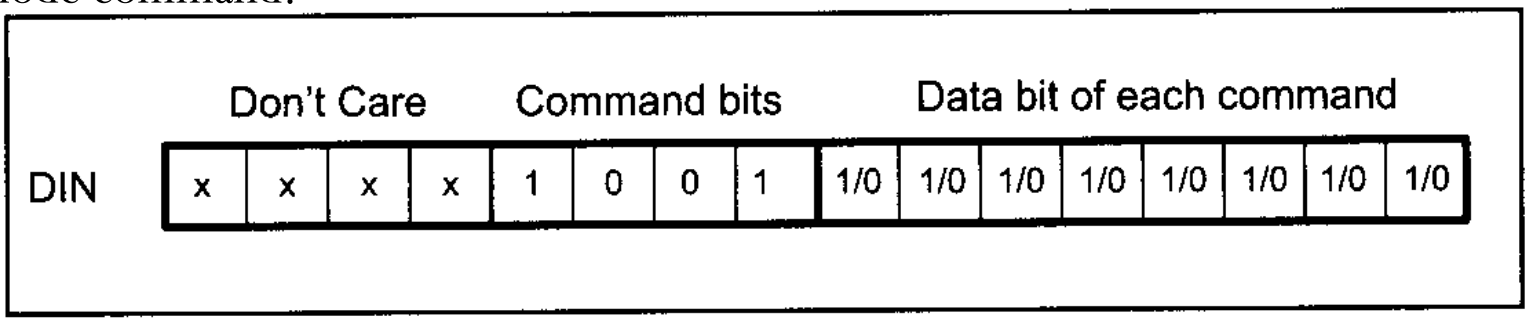


Figure 17-17. Set Decoding Mode Command Format

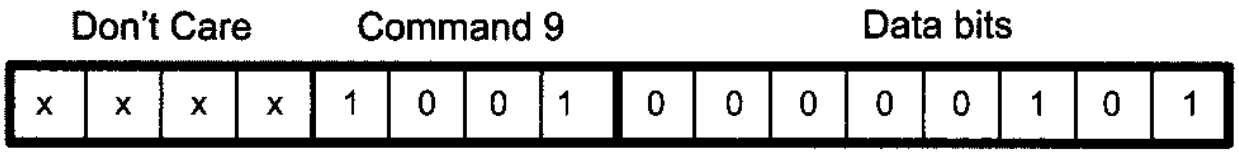
SECTION 17.3 : MAX7221 INTERFACING AND PROGRAMMING

Example 17-5

What sequence of bytes should be sent to the MAX7221 in order to enable the decoding function for digit 0 and digit 2, and disable the decoding function for other digits?

Solution:

The first byte should be xxxx 1001 (X9 hex) to execute the "Set decoding mode" command, and the second byte (argument of the command) should be 0000 0101 to enable the decoding function for digit 0 and digit 2.



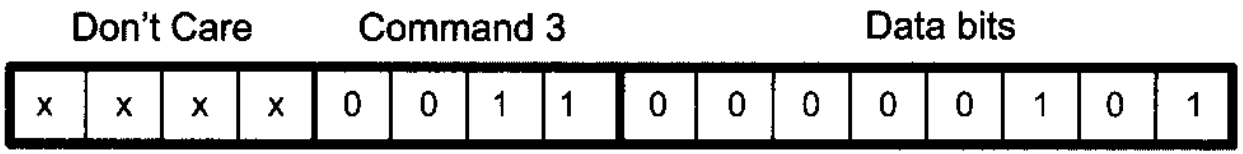
SECTION 17.3 : MAX7221 INTERFACING AND PROGRAMMING

Example 17-6

After running Example 17-5, what sequence of numbers should be sent to the MAX7221 in order to write 5 on digit 2?

Solution:

The first byte should be xxxx 0011 (X3 hex) to execute the "Set value of digit 2" command, and the second byte (argument of the command) should be 0000 0101 (05 hex) to write 5 on digit 2. Notice that the decoding function for digit 2 has been enabled before.



SECTION 17.3 : MAX7221 INTERFACING AND PROGRAMMING

If you want to turn on/off each segment by yourself to display a specific letter on a 7-segment, you should bypass the decoding function and then use the "Set value of digit x" command to turn on/off each bit of a segment. As you see in Figure 17-18, each bit of the data bits is assigned to a segment of the 7-segment. For example, D0 is assigned to the G segment, D1 is assigned to the F segment, and so on. If you want to turn on a segment, you should write one to its bit, and if you want to turn off a segment, you should write zero to its bit. Figure 17-18 shows which are assigned to which segments.

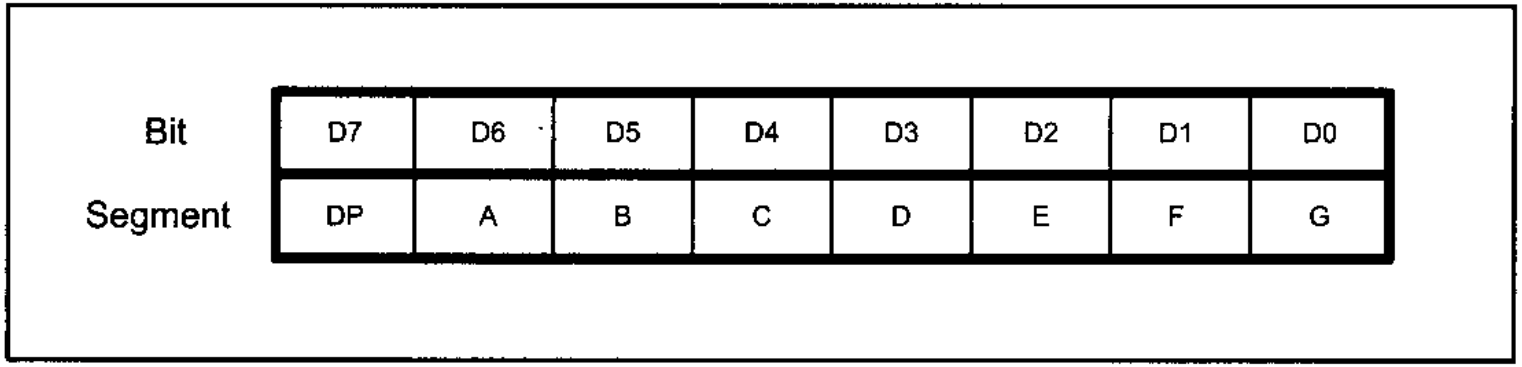
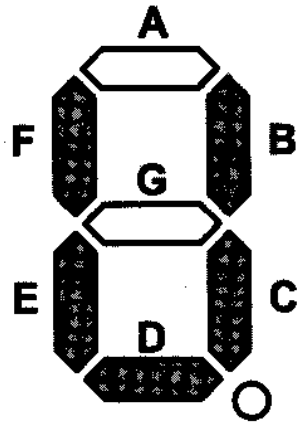


Figure 17-18. Bits Assigned to Segments

SECTION 17.3 : MAX7221 INTERFACING AND PROGRAMMING

Example 17-7

After running Example 17-5, what sequence of numbers should be sent to the MAX7221 in order to write U on digit 1?



Solution:

The decoding function for digit 1 has been disabled before in Example 17-5, and we have to turn on/off each segment manually. As you see in the figure, segments B, C, D, E, and F should be turned on. To turn on these segments of digit 1, we should send the first byte xxxx 0010 (X2 hex) to execute the "Set value of digit 1" command and then we should send 0011 1110 (3E hex) to write U on digit 1. Notice that the decoding function for digit 1 has been enabled before. The figure below shows the bits.

Don't Care				Command 3				Data bits							
x	x	x	x	0	0	1	0	0	0	1	1	1	1	1	0
								DP	A	B	C	D	E	F	G

Set Intensity of Light (command XA)

This command sets the light intensity of the segments. The intensity can be any value between 0 and 16 (0F hex). 0 is the minimum value of intensity, and 16 is the maximum value of intensity. Notice that 0 does not mean off but it is the minimum intensity. As we mentioned before, you can also change the light intensity of segments by changing the resistor that connects the ISET pin to VCC.

Set Scan Limit (command XB)

This command sets the number of 7-segments that are connected to the chip. This number can vary from 1 to 8.

Turn On/ Off (command XC)

This command turns the display on or off. 1 (01 hex) turns the display on, while 0 (00 hex) turns off the display. This command is useful when you want to reduce the power consumption of your device.

Display Test (command XF)

This command is used to test the display. If you send 1 (01 hex) after sending the display test command to the chip, it enters display-test mode and turns on all segments. This lets you check to see if all segments work properly. When you want to return to normal operation mode, you should execute the command but send 0 (00 hex) as data to the chip.

MAX7221 programming in the AVR

To program MAX7221 in the AVR you should do the following steps. Notice that step 4 is optional and can be ignored:

1. Initialize the SPI to operate in master mode 0.
2. Enable or disable decoding mode by executing command 9 (x9 hex).
3. Set the scan limit.
4. Set the intensity of light (optional).
5. Turn on the display.
6. Set the value of each digit.

See Programs 17-1 and 17-2. Program 17-1 shows how to display 57 on the 7-segment display of Figure 17-15 by use of the decoding function. Program 17-2 shows how to display 2U on the 7-segment of Figure 17-15 without using the decoding function.

SECTION 17.3 : MAX7221 INTERFACING AND PROGRAMMING

PA

```
.INCLUDE "M32DEF.INC"

.equ  MOSI = 5
.equ  SCK = 7
.equ  SS = 4

      LDI   R21,HIGH(RAMEND) ;set the high byte of stack
      OUT   SPH,R21          ;pointer
      LDI   R21,LOW(RAMEND)  ;set the low byte of stack
      OUT   SPL,R21          ;pointer

      LDI   R17,(1<<MOSI)|(1<<SCK)|(1<<SS)
      OUT   DDRB,R17         ;MOSI, SCK, and SS are output

      LDI   R17,(1<<SPE)|(1<<MSTR)|(1<<SPR0) ;enable SPI
      OUT   SPCR,R17         ;master mode 0, CLK = fck/16

      LDI   R17,0x09         ;set decoding mode command
      LDI   R18,0b00000011   ;enable decoding for digit 0,1
      CALL  RunCMD           ;send CMD and DATA to the chip
```

SECTION 17.3 : MAX7221 INTERFACING AND PROGRAMMING

```
        LDI   R17,0x0B          ;set scan limit command
        LDI   R18,0x02          ;scan two 7-segments
        CALL  RunCMD            ;send CMD and DATA to the chip

        LDI   R17,0x0C          ;turn on/off command
        LDI   R18,0x01          ;turn on the chip
        CALL  RunCMD            ;send CMD and DATA to the chip

        LDI   R17,0x01          ;select digit 0
        LDI   R18,0x07          ;display value 7
```

Program 17-1: Display 57 on 7-Segment LEDs *(continued on next page)*

SECTION 17.3 : MAX7221 INTERFACING AND PROGRAMMING

P

```
        CALL  RunCMD            ;send CMD and DATA to the chip

        LDI   R17,0x02          ;select digit 1
        LDI   R18,0x05          ;display value 5
        CALL RunCMD             ;send CMD and DATA to the chip

H:      RJMP H                  ;stop here .
;-----------------------------------------------------------
;this function sends a command and its argument (data) to SPI
;command should be in R17 and data should be in R18 before
;the function is invoked
;-----------------------------------------------------------
```

SECTION 17.3 : MAX7221 INTERFACING AND PROGRAMMING

PAGE 609

```
RunCMD:
      CBI  PORTB,SS          ;CS = 0 to start packet
      OUT  SPDR,R17          ;transmit the command in R17
Wait1:
      SBIS SPSR,SPIF         ;skip next instruction if IF is set
      RJMP Wait1             ;otherwise jump to wait1

      OUT  SPDR,R18          ;transmit the data in R18
Wait2:
      SBIS SPSR,SPIF         ;skip next instruction if IF is set
      RJMP Wait2             ;otherwise jump to wait2

      SBI  PORTB,SS          ;CS = 1 to terminate packet
      RET                    ;return
```

Program 17-1: Display 57 on 7-Segment LEDs *(continued from previous page)*

SECTION 17.3 : MAX7221 INTERFACING AND PROGRAMMING

Program 17-2 shows how to display 2U on the 7-segments of Figure 17-15 without using the decoding function.

```
.INCLUDE "M32DEF.INC"

.equ  MOSI = 5
.equ  SCK = 7
.equ  SS = 4

      LDI   R21,HIGH(RAMEND) ;set the high byte of stack
      OUT   SPH,R21          ;pointer
      LDI   R21,LOW(RAMEND)  ;set the low byte of stack
      OUT   SPL,R21          ;pointer

      LDI   R17,(1<<MOSI)|(1<<SCK)|(1<<SS)
      OUT   DDRB,R17         ;MOSI, SCK, and SS are output

      LDI   R17,(1<<SPE)|(1<<MSTR)|(1<<SPR0);enable SPI
```

Program 17-2: Display 2U on the 7-Segment LEDs *(continued on next page)*

SECTION 17.3 : MAX7221 INTERFACING AND PROGRAMMING

P

```
        OUT   SPCR,R17          ;master mode 0, CLK = fck/16

        LDI   R17,0x09          ;set decoding mode command
        LDI   R18,0b00000010    ;enable decoding for digit 1
        CALL  RunCMD            ;send CMD and DATA to the chip

        LDI   R17,0x0B          ;set scan limit command
        LDI   R18,0x02          ;scan two 7-segments
        CALL  RunCMD            ;send CMD and DATA to the chip

        LDI   R17,0x0C          ;turn on/off command
        LDI   R18,0x01          ;turn on the chip
        CALL  RunCMD            ;send CMD and DATA to the chip

        LDI   R17,0x01          ;select digit 0
        LDI   R18,0x3E          ;display U (see Example 17-7)
        CALL  RunCMD            ;send CMD and DATA to the chip

        LDI   R17,0x02          ;select digit 1
        LDI   R18,0x02          ;display 2
        CALL RunCMD             ;send CMD and DATA to the chip

H:      RJMP H                  ;stop here
```

```
;-----------------------------------------------------------
;this function sends a command and its argument (data) to SPI
;command should be in R17 and data should be in R18 before
;the function is invoked
;-----------------------------------------------------------

RunCMD:
      CBI  PORTB,SS          ;CS = 0 to start packet
      OUT  SPDR,R17          ;transmit the command in R17
Wait1:
      SBIS SPSR,SPIF         ;skip next instruction if IF is set
      RJMP Wait1             ;otherwise jump to wait1

      OUT  SPDR,R18          ;transmit the data in R18
Wait2:
      SBIS SPSR,SPIF         ;skip next instruction if IF is set
      RJMP Wait2             ;otherwise jump to wait2

      SBI  PORTB,SS          ;CS = 1 to terminate packet
      RET                    ;return
```

Program 17-2: Display 2U on 7-Segment LEDs *(continued from previous page)*

MAX7221 programming in C

Example 17-8 and Example 17-9 are C versions of Programs 17-1 and 17-2, respectively.

Example 17-8

Write an AVR C program to display 57 on the 7-segments of Figure 17-15.

Solution:

```
#include <avr/io.h>                          //standard AVR header
#define MOSI 5
#define SCK 7
#define SS 4

void execute( unsigned char cmd , unsigned char data)
{
   PORTB &= ~(1<<SS);                        //initializing the packet
                                             //by pulling SS low
   SPDR = cmd;                               //start CMD transmission
   while(!(SPSR & (1<<SPIF)));               //wait transfer finish

   SPDR = data;                              //start DATA transmission
```

SECTION 17.3 : MAX7221 INTERFACING AND PROGRAMMING

P

```
   while(!(SPSR & (1<<SPIF)));          //wait transfer finish

   PORTB |= (1<<SS);                    //terminate the packet by
}                                       //pulling SS high


int main (void)
{
   DDRB = (1<<MOSI)|(1<<SCK)|(1<<SS);  //MOSI and SCK are output
   SPCR = (1<<SPE)|(1<<MSTR)|(1<<SPR0);//enable SPI as master

   execute(0x09,0b00000011);            //enable decoding for
                                        //digits 1,2
   execute(0x0B,0x02);                  //scan two 7-segments
   execute(0x0C,0x01);                  //turn on the chip
   execute(0x01,0x07);                  //display 7
   execute(0x02,0x05);                  //display 5

   while(1);
   return 0;
}
```

F

Example 17-9

Write an AVR C program to display 2U on the 7-segments of Figure 17-15.

Solution:

```
#include <avr/io.h>                          //standard AVR header
#define MOSI 5
#define SCK 7
#define SS 4

void execute( unsigned char cmd , unsigned char data)
{
   PORTB &= ~(1<<SS);                        //initializing the packet
                                             //by pulling SS low
   SPDR = cmd;                               //start CMD transmission
   while(!(SPSR & (1<<SPIF)));               //wait transfer finish

   SPDR = data;                              //start DATA transmission
   while(!(SPSR & (1<<SPIF)));               //wait transfer finish
```

SECTION 17.3 : MAX7221 INTERFACING AND PROGRAMMING

```
   PORTB |= (1<<SS);                    //terminate the packet by
}                                       //pulling SS high


int main (void)
{
   DDRB = (1<<MOSI)|(1<<SCK)|(1<<SS);  //MOSI and SCK are output
   SPCR = (1<<SPE)|(1<<MSTR)|(1<<SPR0);//enable SPI as master

   execute(0x09,0b00000010);       //enable decoding for digit 1
   execute(0x0B,0x02);             //scan two 7-segments
   execute(0x0C,0x01);             //turn on the chip
   execute(0x01,0x3E);             //display U (see Example 17-7)
   execute(0x02,0x02);             //display 2

   while(1);
   return 0;
}
```