SPI Communications

Goal:

- Send and receive serial data using SPI protocol.

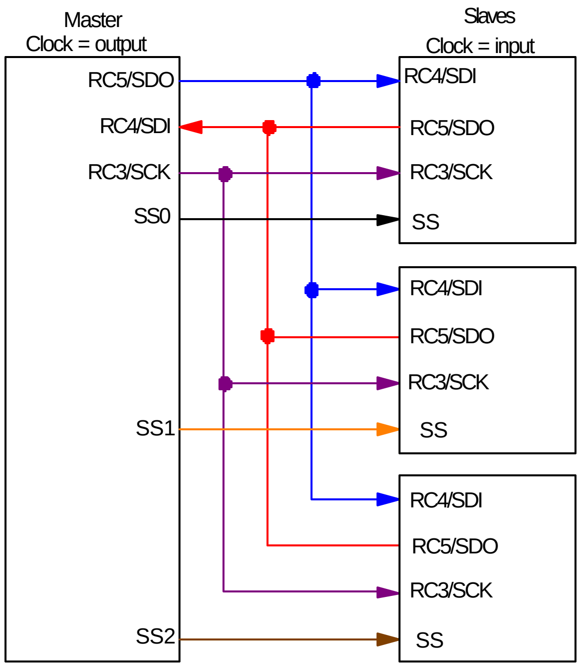
Why:

Efficiency. You can send an unlimited amount of data using 3+ lines:

• CLK:	Clocks the serial data in and out
• SDO	Serial Data Out.
• SDI	Serial Data In
• SS	Slave Select. SS enabled is required for the slave to drive its SDO line.

How: Hardware

- Pick one processor to serve as the Master. This device drives the clock line
- The other devices serve as slaves. These devices also send data on the SDO line and receive on their SDI line, but, they only do so as commanded by the bus master. The master determines when the data is sent (SCK) and which slave is allowed to talk on the slave SDO line (SS).
- You usually need a separate SS (slave select) line for each slave device.



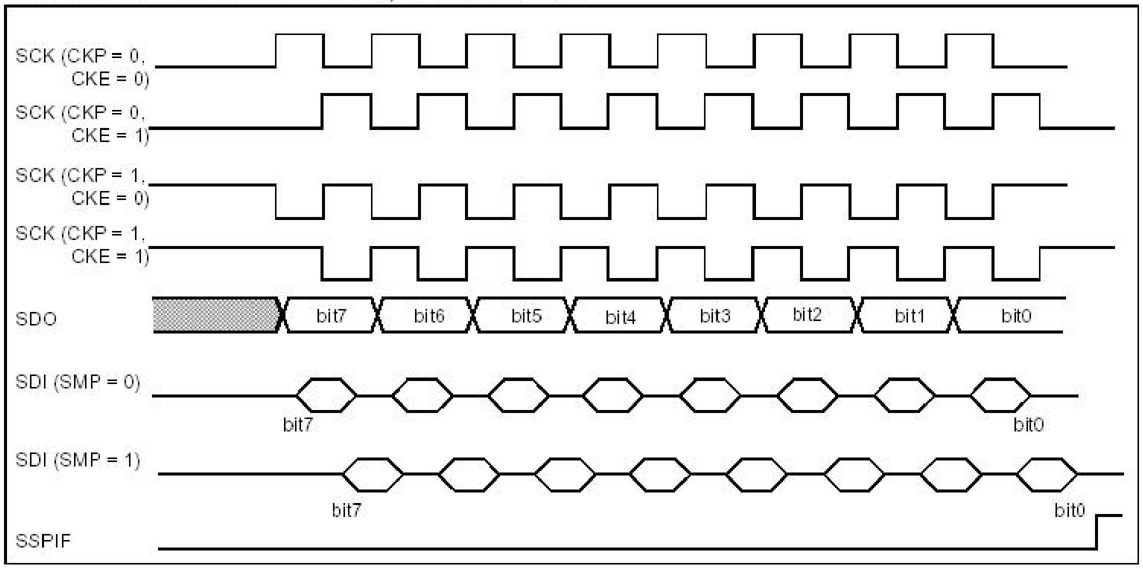
SPI - Master Mode

The bus master controls all communications. The master

- Determines when data is sent / received
- Provides the clock
- Provides the SS line (optional). This determines which slave is active.
- Writes to the SDO line
- Reads the SDI line.

Timing:

FIGURE 9-2: SPI MODE TIMING, MASTER MODE



The master controls the clock line and pulses it once for each bit that is sent. You can adjust the clock's phase and sign using CKP and CKE.

Data is sent out on the SDO line most significant bit first.

At the same time, data is read in on the SDI line, most significant bit first.

After eight bits are sent (one byte), SSPIF is set, signifying that the SPI port is

- ready to be read to see what data the slave sent, and
- ready to send a new byte.

How: Software:

1. Set up PORTC as follows:

TRISC (address 0x__ - Bank __)

Bit	7	6	5	4	3	2	1	0
			SDO	SDI	SPICLK			
input / output	-	-	0	1	0	-	-	-

2. Set up the conditions for the interrupt

SSPSTAT (address 0x94 - Bank 1)

Bit	7	6	5	4	3	2	1	0
Name	SMP	CKE	D/A	P	S	R/W	UA	BF
Value	0	1	x	x	x	x	x	0

SMP: Sample bit:

- SPI Master Mode
  - 1 = Input data sampled at end of data output time
  - 0 = Input data sampled at middle of data output time
- SPI Slave Mode
  - SMP must be cleared when SPI is used in slave mode

CKE: SPI Clock Edge Select

- CKP = 0
  - 1 = Transmit happens on transistion from active clock state to idle clock state
  - 0 = Transmit happens on transistion from idle clock state to active clock state
- CKP = 1
  - 1 = Data transmitted on falling edge of SCK
  - 0 = Data transmitted on rising edge of SCK

BF: Buffer Full Status bit

- Receive (SPI and I2C modes)
  - 1 = Receive complete, SSPBUF is full
  - 0 = Receive not complete, SSPBUF is empty

SSPCON (address 0x14 - Bank 0)

Bit	7	6	5	4	3	2	1	0
Name	WCOL	SSPOV	SSPEN	CKP	SPI Master Clock Freq			
Value	0	0	1	0	0	0	a	b

WCOL: Write Collision Detect bit

- Master Mode:
  - 1 = A write to SSPBUF was attempted while the I2C conditions were not valid
  - 0 = No collision
- Slave Mode:
  - 1 = SSPBUF register is written while still transmitting the previous word (must be cleared in software)
  - 0 = No collision

SSPOV: Receive Overflow Indicator bit

- 1 = A new byte is received while SSPBUF holds previous data. Data in SSPSR is lost on overflow.
  - In slave mode the user must read the SSPBUF, even if only transmitting data, to avoid overflows.
  - In master mode the overflow bit is not set since each operation is initiated by writing to the SSPBUF register. (Must be cleared in software).
- 0 = No overflow

SSPEN: Synchronous Serial Port Enable bit. In SPI mode, when enabled, these pins must be properly configured as input or output.

- 1 = Enables serial port and configures SCK, SDO, SDI, and SS as the source of the serial port pins
- 0 = Disables serial port and configures these pins as I/O port pins

CKP: Clock Polarity Select bit

- 1 = Idle state for clock is a high level
- 0 = Idle state for clock is a low level

bit 3-0: SSPM3:SSPM0: Synchronous Serial Port Mode Select bits

- 0000 = SPI master mode, clock = FOSC/4
- 0001 = SPI master mode, clock = FOSC/16
- 0010 = SPI master mode, clock = FOSC/64
- 0011 = SPI master mode, clock = TMR2 output/2
- 0100 = SPI slave mode, clock = SCK pin. SS pin control enabled.
- 0101 = SPI slave mode, clock = SCK pin. SS pin control disabled. SS can be used as I/O pin

SSPSM3:SSPSM0	Clock	with a 20MHz crystal
00000	$F_{OSC}/4$	5MHz
0001	$F_{OSC}/16$	1.25MHz
0010	$F_{OSC}/64$	312.5kHz

Note that SPI communications is very fast: up to 5 million bits / second being transferred each way.

3. Enable an INT interrupt
   - SSPIE = 1

4. Enable all interrupts:
   - PEIE = 1;
   - GIE = 1: enable all interrupts

At this point, you're ready to send and receive 8 bits of data using the on-board SPI port.

Example: Write a routine which

- Passes 8 bits on SDO and
- Returns the 8-bits received on SDI
- Determine how long it takes to send data at the maximum bit rate

Calling Format:

```
Result = SPI_RW('I');                       // 'I' is sent on SDO
                                            // the data from SDI is
                                            // returned in Result
```

Subroutines:

```
void Init_SPI(void)
{
   SSPCON = 0x20;                           // SPI is on, 5Mbps
   TRISC5 = 0;                              // set up PORTC for SPI
   TRISC4 = 1;
   TRISC3 = 0;
   SSPIE = 0;                               // turn off interrupts
                                            // use polling instead
   }


unsigned char SPI_RW(unsigned char Data)

{
   unsigned char Result;

   SSPIF = 0;
   SSPBUF = Data;                           // load the SPI data buffer
   while (!SSPIF);                          // wait until data sent
   Result = SSPBUF;                         // load what the slave sent

   return(Result);                          // and return it
   }
```

To send 8 bits, this routine takes

- $(8 \text{ bits})\left(\frac{1}{5\text{Mbps}}\right) = 1.6uS$ to send the data
- + overhead to call the subroutine, read a character, and return a character.

SPI communication is a convenient and fast way to send data from one chip to another on the same circuit board.

SPI - Slave

SPI slave mode is just like the master mode, save the clock line is driven by someone else (CLK = input for slaves). If using more than one slave, tie the slave select lines from the master to RA5. This causes the slave to ignore the data unless SS=0. It also synchronizes the data to the falling edge of SS.

Note: Since you can't control how fast the data is sent, the slave often times

- Uses an interrupt to receive each byte (you don't have time to wait since the next byte may be coming within the next 1.6us)
- Save each byte in a stack for temporary storage (you don't have time to decode the data as it comes in. The next byte may be coming in the next 1.6us).

The main routine can then figure out what the data means and what it's supposed to to with this data once it's been received.

Timing:

FIGURE 9-3: SPI MODE TIMING (SLAVE MODE WITH CKE = 0)

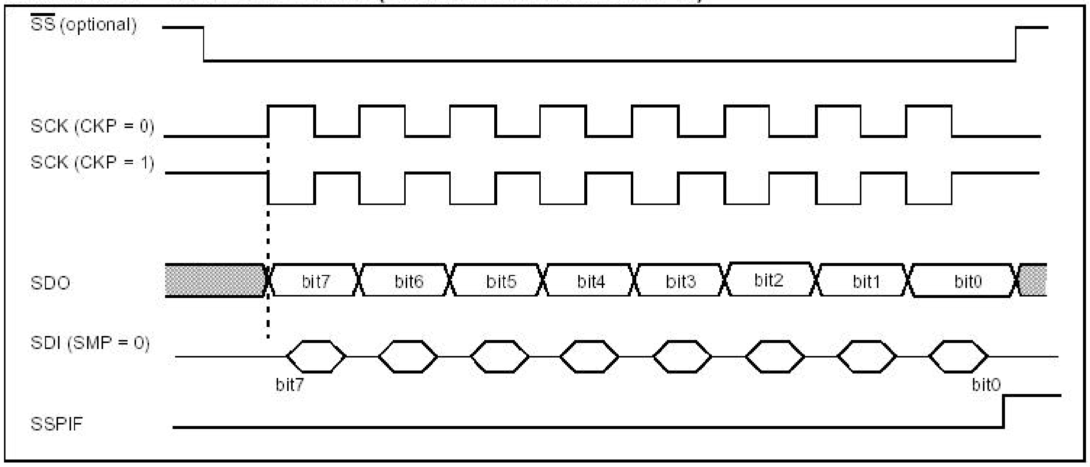
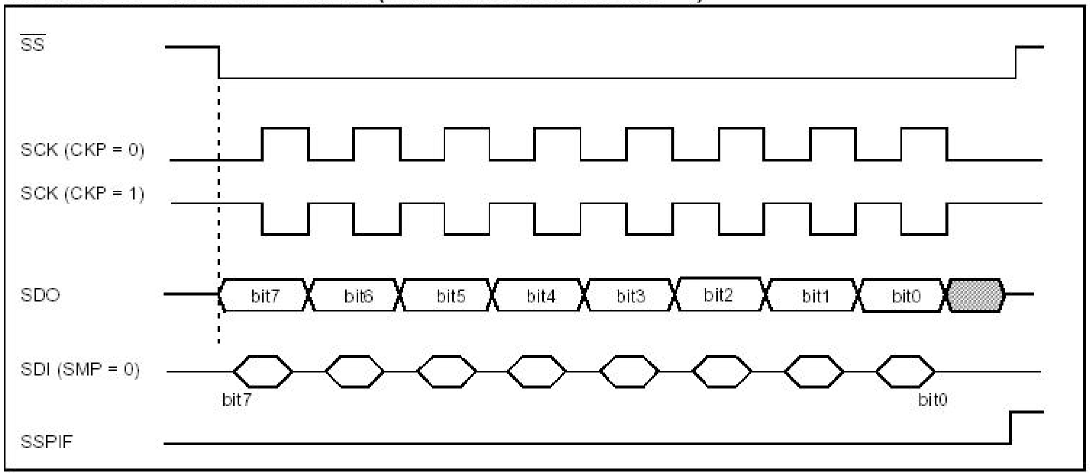


FIGURE 9-4: SPI MODE TIMING (SLAVE MODE WITH CKE = 1)



How: Software:

1. Set up PORTC as follows:

TRISC (address 0x__ - Bank __)

Bit	7	6	5	4	3	2	1	0
			SDO	SDI	SPICLK			
input / output	-	-	0	1	0	-	-	-

2. Set up the conditions for the interrupt

SSPCON (address 0x14 - Bank 0)

Bit	7	6	5	4	3	2	1	0
Name	WCOL	SSPOV	SSPEN	CKP	SPI Slave			
Value	0	0	1	0	0	1	0	a

- a = 0: SS disabled (SS is pin RA5)
- a = 1: SS enabled. (RA5)

3. Enable an SPI interrupt (The slave really needs to use interrupts since it can't control how fast the data comes in. The slave needs to read the data and save it as fast as possible.)

- SSPIE = 1

4. Enable all interrupts:

- PEIE = 1;
- GIE = 1: enable all interrupts

Example Code:

1. Initialization Routine for SPI Slaves:

```
void Init_SPI_Slave(void)
{
   TRISA5 = 1;                           // RA5 = Slave Select
   SSPCON = 0x25;                        // turn on SPI slave mode
   SSPIF = 0;
   SSPIE = 1;                            // enable interrupts
   PEIE = 1;
   GIE = 1;
   }
```

2. Interrupt Service Routine:

- In this example, load the data which is incoming into a buffer, SPI_STACK, as it is read in.
- Assume that no data is sent back to the master.

Interrupt Service Routine: Read the SPI port and place the data into a circular stack of six

```
// Global
   unsigned char Stack_Pointer;
   unsigned char SPI_Stack[6];

// Interrupt Service Routine

void interrupt IntSer(void) @ 0x10
{
   if (SSPIF == 1) {                     // got a byte
      SPI_Stack[Stack_Pointer] = SSPBUF;
      Stack_Pointer += 1;
      Stack_Pointer = Stack_Pointer % 6;

//    SSPBUF = DATA;                     // if you are going to
//                                       // return data, do it here

      SPIF = 0;
      }
   }
```

Example: DS1267 Digital Potentiometer: Write a routine which sets the value of a DS1267 digital potentiometer. Make the calling function

```
     void Set_Pot(unsigned char A, unsigned char B)
```

where A and B are the values of the two potentiometers: $R = R_0\left(\frac{A}{255}\right)$.

Solution:

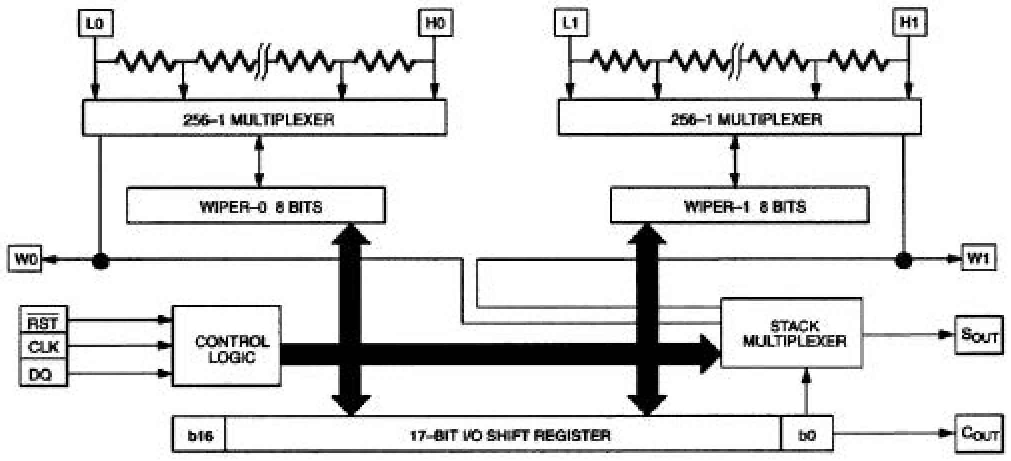
Step 1. Find the data sheets for a DS1267.

Step 2. Set up the hardware connections. Connect

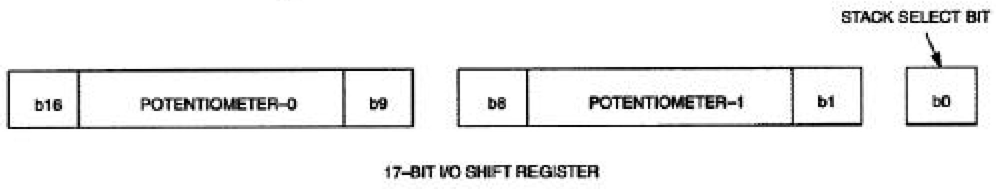
- DQ to the SPI data line (SDO)
- CLK to the SPI clock line (SCK)
- RST to some other pin on the PIC chip (sort of a slave select line.) Assume RC0 for now.

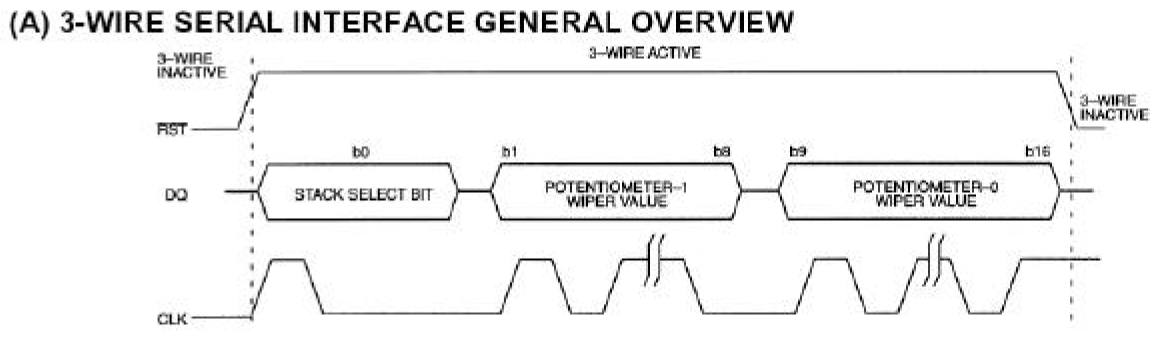
Step 3. In the data sheets, find a timing diagram. This is given in Figure 9 and Figure 1:

DS1267 BLOCK DIAGRAM Figure 1



I/O SHIFT REGISTER Figure 2




(A) 3-WIRE SERIAL INTERFACE GENERAL OVERVIEW
3-WIRE INACTIVE
3-WIRE ACTIVE
3-WIRE INACTIVE
RST
DQ
CLK
b0
b1
b8
b9
b16
STACK SELECT BIT
POTENTIOMETER-1 WIPER VALUE
POTENTIOMETER-0 WIPER VALUE

From these diagrams, it appears you need to...

- Send 24 bits (17 bits, rounded up to a multiple of eight)
- This data is shifted through a shift register on the DS1267 so that you can cascade several digital pots. The last 17-bits send are the ones that matter. These 24-bits should look like the following:
- Flip the bits in each byte. The data bits are received LSB first for the DS1267 while the PIC sends the data MSB first. Either write your own SPI routine or write a flip routine.

First byte sent								Second Byte Sent								Third Byte Sent							
x	x	x	x	x	x	x	0	b0	b1	b2	b3	b4	b5	b6	b7	b0	b1	b2	b3	b4	b5	b6	b7
padding bits and Stack Select								Pot 1 value - LSB first								Pot #2 value - LSB first							

Step 4. Write some code:

Option #1: Use the On-Chip SPI routine. Assume Y=flip(X) flips all bits:

```
void Set_Pot(unsigned char A; unsigned char B)
{
static bit RST  @ ((unsigned)&PORTC*8+0);  // RST = RC0

   RST = 0;
   RST = 1;                                // select the DS1267

   SSPIF = 0;
   SSPBUF = 0;
   while (!SSPIF);                         // send first byte

   SSPIF = 0;
   SSPBUF = flip(A);
   while (!SSPIF);                         // send second byte

   SSPIF = 0;
   SSPBUF = flip(B);
   while (!SSPIF);                         // send third byte

   RST = 0;                                // deselect the DS1267
}
```

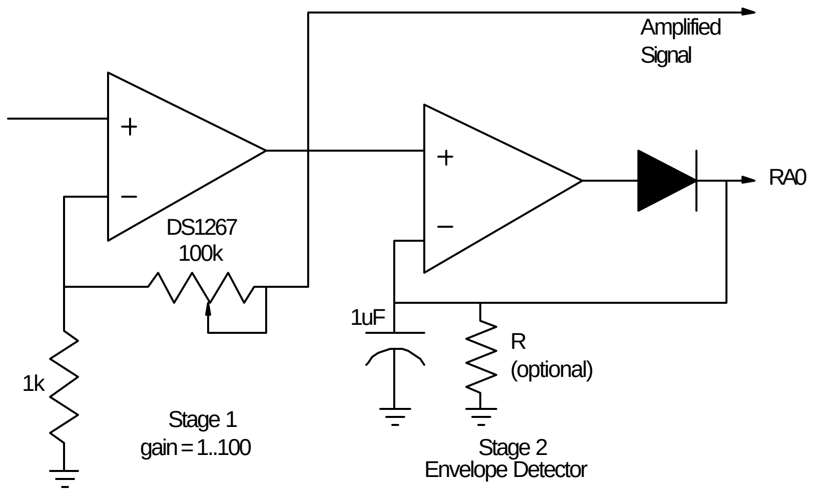
Option #2: Write your own SPI port driver:

```
static bit RST  @ ((unsigned)&PORTC*8+0);
static bit DQ   @ ((unsigned)&PORTC*8+5);
static bit CLK  @ ((unsigned)&PORTC*8+3);

void Set_Pot(unsigned char A; unsigned char B)
{
   unsigned char i;

   RST = 0;                                 // default levels for RST
   SCK = 0;                                 // and clock

   RST = 1;                                 // select the DS1267

   DQ = 0;                                  // Stack select = 0
   SCK = 1;   SCK = 0;                      // pulse the clock

   for (i=0; i<8; i++) {
      if (((A>>i) & 1) == 0) DQ = 0;        // send each bit of A
                        else DQ = 1;        // LSB first
      SCK = 1;  SCK = 0;
      }

   for (i=0; i<8; i++) {                    // send each bit of B
      if (((B>>i) & 1) == 0) DQ = 0;        // LSB first
      else DQ = 1;
      SCK = 1;
      SCK = 0;
      }

   RST = 0;                                 // after 17 bits, you're done
   }
```

What can you do with a digital pot?

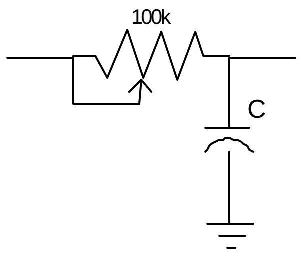
1) Design an amplifier which the PIC can monitor and adjust.

The PIC monitors the amplitude of the signal at RA0. If it is too small, increases the gain (increase R). If it is too large and is saturating the amplifier (RA0 too close tp +5V), reduce the gain (reduce R).

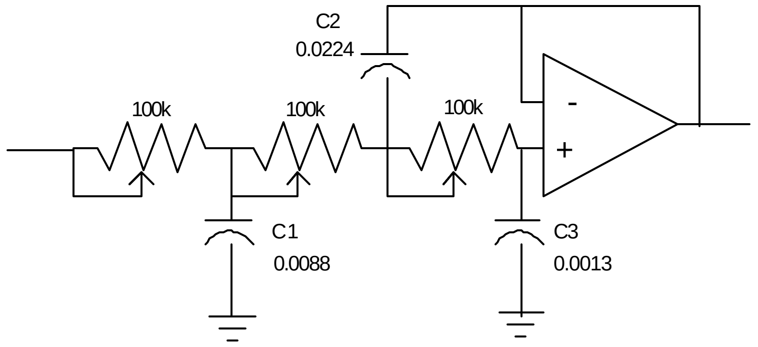


2) Design a tunable low-pass filter. Set the corner from 100Hz to 1kHz.

By adjusting R, you set the corner frequency. A 1-stage filter is shown on the left. A 3-stage Butterworth filter is shown on the right. All three R's should have the same value.



1-Stage Butterworth Filter



3-Stage Butterworth Filter

max corner = 0 (R=0) (from ECE 321 text)

min corner = $2\pi f = \left(\frac{1}{100k \cdot C}\right)$

2-Wire SPI Communications:

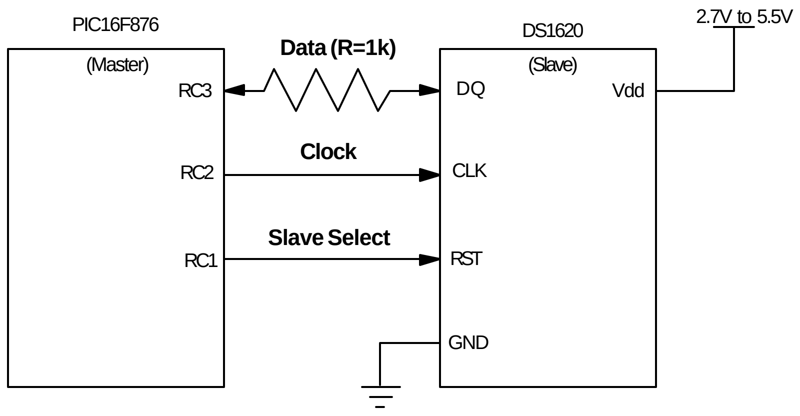
Some companies, such as Dallas Semiconductors, use a 2-wire SPI communication scheme. This uses

- A clock line from the master (as a normal SPI communication scheme), but
- A single data line.

The concept is that often times you only have communication going one way. If you constrain all communications to happen in one direction only (termed half duplex), you can save one data line:

- When the master is talking, it drives the data line and the slave listens.
- When the slave replies, the master switches its data line to input (high z) and the slave drives the data line.

Since the data can go both ways, add a 1k resistor in-between as a buffer. This limits the current flow if the PIC and the DS1620 try to drive the data line at the same time. This *should* never happen, if the program is written properly. It doesn't hurt to be safe, though.



Step 1. Find the data sheets for a DS1620

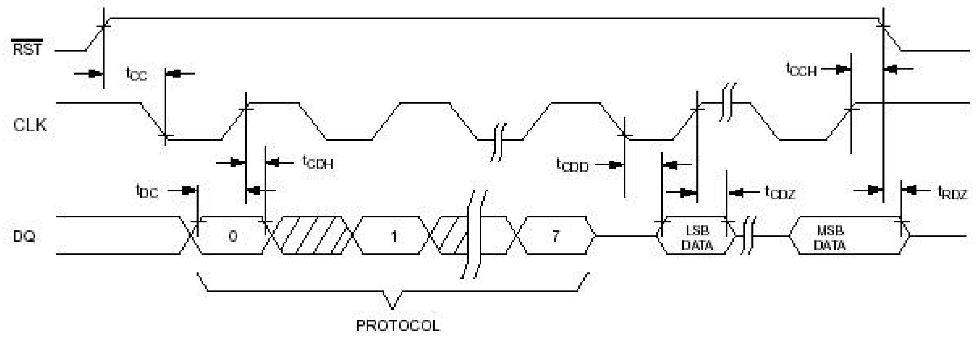
Step 2. Find the hardware connections. Arbitrarily, let

- RC1: RST (1 = start comm. 0 = terminate)
- RC3: DQ (1k resistor between)
- RC2: CLK (pulse low = clock)

Step 3. Find the timing (figure 4 from the data sheets):

DS1620

READ DATA TRANSFER Figure 4



You

- Start with CLK = 1, RST = 0.
- Pull RST high to start communications
- Set DQ = output for the PIC to drive the data line.
- Clock out 8 bits of data with the data valid on the rising edge of the clock. This is the command for the DS1620 processor.
- If you are sending data to the DS1620, keep DQ as output. Clock out 8 more bits, LSB first.
- If you are receiving data, switch DQ to input. Clock in 9 bits of data, LSB first. Data valid on clock low (not obvious on the data sheets but found experimentally in lab)

Step 4. Find in the data sheets what commands you need to use to start a temperature conversion, read temperature, etc.

DS1620 COMMAND SET Table 3

INSTRUCTION	DESCRIPTION	PROTOCOL	3–WIRE BUS DATA AFTER ISSUING PROTOCOL	NOTES
TEMPERATURE CONVERSION COMMANDS				
Read Temperature	Reads last converted temperature value from temperature register.	AAh	<read data>	
Read Counter	Reads value of count remaining from counter.	A0h	<read data>	
Read Slope	Reads value of the slope accumulator.	A9h	<read data>	
Start Convert T	Initiates temperature conversion.	EEh	Idle	1
Stop Convert T	Halts temperature conversion.	22h	Idle	1
THERMOSTAT COMMANDS				
Write TH	Writes high temperature limit value into	01h	<write data>	2

Example:

Write routines to

- Define the bits used in a more meaningful way that RC0...
- Send 8 bits to the DS1620
- Receive 9 bits from the DS1620
- Initialize the DS1620 to continuous temperature readings, and
- Read the temperature

a) Define the bits in a more meaningful way:

```
static bit DQ      @ ((unsigned)&PORTC*8+3);        // DS1620 - DQ
static bit CLK     @ ((unsigned)&PORTC*8+2);        // DS1620 - CLK
static bit RST     @ ((unsigned)&PORTC*8+1);        // DS1620 - RST
```

b) A subroutine to send 8 bits to the DS1620:

```
void DS1620_Write(unsigned char Data)
{
   unsigned char i;

   TRISC1 = 0;                               // set up data pin I/O
   TRISC2 = 0;                               // The PIC drives the data
   TRISC3 = 0;                               // line here

   CLK = 1;                                  // default is CLK=1 for the
   for (i=1; i<=8; i++) {                    // DS1620
      DQ = (Data & 1);                       // Ship out each bit
      CLK = 0;                               // LSB first and pulse the
      CLK = 1;                               // clock line
      Data = Data >> 1;
      }
   }
```

c) A subroutine to read 9 bits from the DS1620

```
int DS1620_Read(void)
{
   unsigned int Data;
   unsigned int Temp;
   unsigned char i;

   TRISC1 = 0;
   TRISC2 = 0;
   TRISC3 = 1;                               // here, DQ is driven by the
   CLK    = 1;                               // DS1620
   Data   = 0;
   Temp   = 1;

   for (i=1; i<=9; i++) {                    // read in 9 bits
      CLK = 0;                               // data valid when CLK=0
```

```
         if (DQ == 1) Data += Temp;           // data comes in LSB first
         Temp = Temp * 2;
         CLK = 1;
         }
      return(Data);                           // after 9 bits return Data
      }
```

A Subroutine to initialize the DS1620 to continuous conversion mode

```
   void DS1620_Init(void)
   {
      RST = 0;                                // Start communications

      RST = 1;                                // with RST going high
      DS1620_Write(0x0C);                     // x0C = Write Config Reg = 00
      DS1620_Write(0x00);                     // (sets up continuous conv)
      RST = 0;

      RST = 1;
      DS1620_Write(0xEE);                     // xEE = Start T conversion
      RST = 0;

      }
```

A subroutine to read the temperature

```
   int DS1620_Read_Temperature(void)
   {
      int Data;
      RST = 0;

      RST = 1;                                // Start communications
      DS1620_Write(0xAA);                     // xAA = read T
      Data = DS1620_Read();                   // next 9 bits = temp reading
      RST = 0;

      return(Data);
      }
```