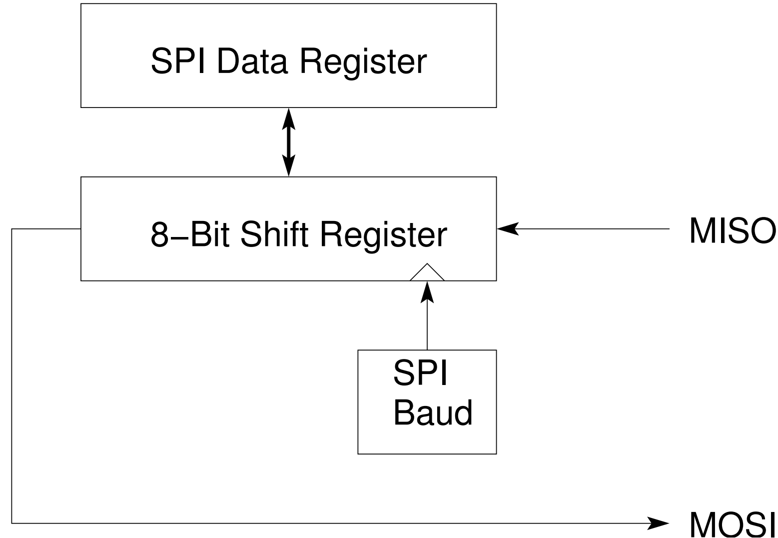
Using the 9S12 SPI

- The SPI has a data register (SPIDR) and a shift register.
- To write data to the SPI, you write to the SPIDR data register. The 9S12 automatically transfers the data to the shift register for transfer.
  - The SPTEF (SPI Tranmsit Empty Flag) indicates whether it is okay to write new data to the SPIDR.
  - Make sure the SPTEF is set before writing to the SPI Data Register.
  - The SPTEF flag is automatically cleared when data is written to the SPI data register, and is automatically set when data is transfered from the data register to the shift register.
  - When you first write a byte of data to SPIDR, the SPTEF bit is cleared, the data is transfered immediately to the shift register, and the SPTEF flag is set, indicating that it is okay to write another byte to SPIDR. It does not mean that the transmission of the first byte has finished.



- To read data from the SPI, you read from the SPIDR after the transmission has completed.
  - The SPIF (SPI Flag) indicates that a tranmission has completed, and that it is okay to read data from the SPIDR.

– Make sure the SPIF (SPI Flag) is set before reading from the SPI Data Register.
– The SPIF flag is automatically cleared by reading the SPISR followed by reading the SPIDR.

Configuring the SPI

1. Enable SPI (`SPE` bit of `SPI0CR1`)
2. Clock phase and polarity set to match device communicating with
   (a) Select clock polarity – `CPOL` bit of `SPI0CR1`
   - `CPOL` = 0 for clock idle low
   - `CPOL` = 1 for clock idle high

   (b) Select clock phase – `CPHA` bit of `SPI0CR1`
   - `CPHA` = 0 for data valid on first clock edge
   - `CPHA` = 1 for data valid on second clock edge
3. Select master or slave `MSTR` bit of `SPI0CR1`
   - Will be master when talking to devices such as D/A, A/D, clock, etc.
   - May be slave if talking to another microprocessor
4. If you want to receive interrupt after one byte transfered, enable receive interrupts by setting `SPIE` bit of `SPI0CR1`
   - Normally master will not use interrupts – transfers are fast enough that you will normally wait for transfer to complete
   - Will often use interrupts when configured as a slave – you will get interrupt when master sends you data
5. If you want to receive interrupt when okay to transmit next byte, enable transmit ready interrups by setting `SPTIE` bit of `SPI0CR1`
6. Configure `LSBFE` of `SPI0CR1` for MSB first (`LSBFE = 0`) or LSB first (`LSBFE = 1`)
   - For most devices, use MSB first

7. Configure for normal or bidirectional mode
   (a) Configure for normal mode by clearing bit `SPC0` of `SPI0CR2`
   (b) Configure for bidirectional mode by setting bit `SPC0` of `SPI0CR2`
   - In Bidirectional Mode, use Bit BIDIROE of SPI0CR2 to control the direction of the data line
   - BIDIROE = 1 `=>` data line output
   - BIDIROE = 0 `=>` data line input

Using the 9S12 SPI in Master Mode

- When using the 9S12 SPI in Master Mode you can talk to several devices over the same serial lines
- When sending data to a device:
  1. Select the device (usually be setting a general purpose I/O line active)
     - If using bidirectional mode, make data line an output by setting BIDIROE
  2. The SPTEF (SPI Transmit Empty Flag) must be set before writing to the SPI data register. Keep reading the SPISR register until SPTEF becomes set.
  3. Write the data to the SPI data register (starts the serial clock, sends data out the MOSI pin and brings data in from MISO pin)
  4. Repeat steps 2, and 3 if more data to send
  5. Deselect the device (usually be making a general purpose I/O line inactive) **after** tranmission is completed.
     - The SPIF flag indicates transmission is complete. You should wait until SPIF is set before deselecting the slave
     - However, the SPIF flag may have been set before the 9S12 started sending the final byte of the sequence.
     - The simplest way to make sure you do not deselct the slave before the last transmission is finished is to always wait for the SPIF flag before sending a new byte to the data register, then reading the data register to clear the SPIF flag.

The following assumes that the SPI has been set up in master mode, and Bit 7 of PORT S has been made a general purpose output pin.

```
PTS = PTS & ~0x80;                  /* Bring SS low to select device */

while ((SPI0SR & 0x20) == 0) ;      /* Wait until SPTEF flag set */
SPI0DR = data[0];                   /* Write data to device */
while ((SPI0SR & 0x80) == 0) ;      /* Wait until transmission complete */
tmp = SPI0DR;                       /* Read data register to clear SPIF */
```

```
while ((SPI0SR & 0x20) == 0) ;     /* Wait until SPTEF flag set */
SPI0DR = data[1];                  /* Write data to device */
while ((SPI0SR & 0x80) == 0) ;     /* Wait until transmission complete */
tmp = SPI0DR;                      /* Read data register to clear SPIF */

PTS = PTS | 0x80;                  /* Bring SS high to deselect device */
```

Using the 9S12 SPI in Master Mode

- When reading data from a device:
  1. Select the device (usually be bringing a general purpose I/O line active)
     - If using bidirectional mode, make data line an input by clearing BIDIROE
  2. The SPTEF (SPI Transmit Empty Flag) must be set before writing to the SPI data register. Read the SPISR register until SPTEF becomes set.
  3. Write a junk byte to the SPI data register (starts the serial clock, which brings data in from MISO)
  4. Wait until the SPIF flag is set (which indicates data transfer is finished) by reading the SPI status register until the SPIF bit is 1 (this is also the first step in clearing the SPIF)
  5. Read data from the SPI data register (gets data and completes clearing the SPIF flag)
  6. Repeat steps 2, 3, 4 and 5 if more data to receive
  7. Deselect the device (usually be bringing a general purpose I/O line inactive)

The following assumes that the SPI has been set up in master mode, and Bit 7 of PORT S has been made a general purpose output pin.

```
PTS = PTS & ~0x80;             /* Bring SS low to select device */

while ((SPI0SR & 0x20) == 0) ; /* Wait until SPTEF flag set */
SPI0DR = 0;                    /* Write junk byte to device */
while ((SPI0DR & 0x80)==0) ;   /* Wait for transfer to finish */
data[0] = SPI0DR;              /* Get data byte; clears SPIF */

while ((SPI0SR & 0x20) == 0) ; /* Wait until SPTEF flag set */
SPI0DR = 0;                    /* Write junk byte to device */
while ((SPI0DR & 0x80)==0) ;   /* Wait for transfer to finish */
data[0] = SPI0DR;              /* Get data byte; clears SPIF */

PTS = PTS | 0x80;              /* Bring SS high to deselect device */
```

Using the 9S12 SPI in Slave Mode

- When using the 9S12 SPI is Slave Mode you can send and/or receive data only when the master device brings your Slave Select line low
- You know when a transfer is complete when the SPIF flag is set
- When using the 9S12 SPI in Slave Mode you often use interrupts so you respond to a transfer in an interrupt service routine
- If you need to send data to the master SPI, you need to put that data in SPI0DR *before* the master starts the data transfer
- You may need to use a general purpose I/O line to let the master know you have data to send it
- Example: Receiving data from an SPI master:
  1. Wait for SPIF flag to become set — this happens when master device brings your slave select low, and sends 8 ticks on the serial clock
  2. Read the SPI0DR to get data from master

```
while ((SPI0SR & 0x80)==0) ;   /* Wait for transfer to finish */
data = SPI0DR;                  /* Get data byte; clears SPIF */
```

The Dallas Semiconductor (Maxim) DS1302 Real Time Clock

- In a future lab you will connect a DS1302 Real Time Clock to your 9S12 board. A copy of the data sheet for the DS1302 is available from the same location you got these notes, or you can download it from the Maxim website at www.maxim-ic.com.
- Be sure to have a copy of the data sheet when reading these notes.
- As outlined in the General Description part of the DS1302 data sheet. the DS1302 is:
  - A Real Time Clock which keeps track of time and date, and is correct for leap years.
  - It uses a simple 3-wire interface
- To use the DS1302 with the 9S12 you need to know how to make the hardware connections, and how to set up the 9S12 software protocol to be compatible with the DS1302
- As shown in the data sheet, the DS1302 has an active high chip enable (`CE`) input (which can be connected to a general purpose I/O pin of the 9S12, or to the `SS` line of the 9S12 SPI interface), a SCLK line (which should be connected to the 9S12's SCK line) and a bidirectional `I/O` line (which should be connected to the 9S12's MOMI line)
- The DS1302 has two inputs for a 32.764 kHz crystal. The DS1302 divides the 32.764 kHz crystal by 2^{15} to generate a one second pulse for time-keeping.
- The DS1302 has power V_{CC2} and ground `GND` inputs. V_{CC2} can range from 2 to 5.5 V.
- The DS1302 has backup power V_{CC1} input. This can be connected to a battery to keep time when power is turned off.

Using the DS1302 Real Time Clock

- The serial transfer protocol shown in Figure 5 of the data sheet shows the following:
  - `CE` is active high
  - The serial clock is idle low, so you should use CPOL = 0
  - The data is valid on the first clock edge, so you should use CPHA = 0
  - The data is sent out with Least Significant Bit (LSB) first, so you should use LSBFE = 1
  - The 9S12 needs to transfer multiple bytes with `CE` high, so you must use SSOE = 0
- The General Description says the the DS1302 will work with a serial clock speed of up to 2 MHz (when the supply voltage is 5 V). Further details of this are given in the AC Electrical Characteristics table on Page 11.

Using the DS1302 Real Time Clock with the 9S12

Writing data to the DS1302

- To write data to the DS1302 you need to send a control byte followed by a data byte, as shown in Figure 5
- The control byte format is shown in Figure 4, with an example in Table 3. The 8 control bits work as follows:
  - Bit 7 must be 1
  - Bit 6 is a 0 to access the clock, a 1 to access the 31 bytes of RAM
  - Bits 5 through 1 indicate which register to access
  - Bit 0 is a 1 to read from the DS1302, and a 0 to write to the DS1302
- The DS1302 uses 8 registers to access the clock, and 1 register to access a trickle charge function (to charge a rechargeable battery), as shown in Table 2.
  - Note: Bit 7 of the seconds register is a Clock Halt bit — writing a 1 to bit 7 of the seconds register stops the oscillator, and puts the DS1302 into a low power mode.

    When setting the time, remember to write a 1 to Bit 7 of the seconds register to start the oscillator.
  - Note: Bit 7 of register 8 is a Write Protect bit. When high you cannot write to the clock registers or RAM.

    In order to set the clock, you must write a 0 the the Write Protect bit.

    After setting the clock, you might want to bring the Write Protect bit high to
- You can write the time by writing one register at a time — make MOMI an output, bring `CE` high, write a control word to select the appropriate register, write the data to the register, bring `CE` low.
- You can use burst mode to write all the register in succession — make MOMI an output, bring `CE` high, write the control word `0xBE` to select register 31, write the time to the registers in order seconds, minutes, hours, day of month, month, day of week, year, then bring `CE` low.

A C program to use the DS1302 with the 9S12 SPI

```
/* Program to write the seconds register of the DS1302
 * It is assumed that the CE line of the DS1302 is connected to
 * the SS line of SPI0 of the 9S12 */
#include "hcs12.h"

main()
{
    short tmp;                    /* Temporary variable */

    DDRS = DDRS | 0x80;           /* SS output */
    PTS = PTS & ~0x80;            /* Deselect DS1302 */

    SPI0CR1 = 0x51;  /* 0 1 0 1 0 0 0 1
                        | | | | | | | |
                        | | | | | | | \____ LSB first
                        | | | | | | \______ multiple bytes with SS asserted
                        | | | | | \________ 0 phase (data valid on 1st edge)
                        | | | | \__________ 0 polarity (clock active low)
                        | | | \____________ Master mode
                        | | \______________ No interrupt on transmit reg empty
                        | \________________ Enable SPI
                        \__________________ No interrupt on transfer complete
                    */

    SPI0CR2 = 0x01;  /* 0 0 0 0 0 0 0 1
                        | | | | | | | |
                        | | | | | | | \____ Bidirectional mode
                        | | | | | | \______ SPI operates normally in wait mode
                        | | | | | \________ Not used
                        | | | | \__________ MOMI line input
                        | | | \____________ Disable MODF flag
                        | | \______________ Not used
                        | \________________ Not used
                        \__________________ Not used
                    */


    SPI0BR = 0x50;             /* 2 MHz SPI clock */
```

```
    PTS = PTS | 0x80;               /* Select DS1302 */

    SPI0CR2 = SPI0CR2 | 0x08;       /* Make MOMI an output */

    while ((SPI0SR & 0x20) == 0) ;  /* Wait until okay to transmit */
    SPI0DR = 0x80;                  /* Select Seconds register for write */
    while ((SPI0SR & 0x80) == 0) ;  /* Wait until transmission complete */
    tmp = SPI0DR;                   /* Read data register to clear SPIF */

    while ((SPI0SR & 0x20) == 0) ;  /* Wait until okay to transmit */
    SPI0DR = 0x35;                  /* Set Seconds to 35, enable oscillator */
    while ((SPI0SR & 0x80) == 0) ;  /* Wait until transmission complete */
    tmp = SPI0DR;                   /* Read data register to clear SPIF */

    PTS = PTS & ~0x80;              /* Deselect slave */
    asm(" swi");
}
```

Using the DS1302 Real Time Clock with the 9S12

Reading data from the DS1302

- To read data to the DS1302 you need to send a control byte followed by a data byte, as shown in Figure 5
- Bit 0 the the control byte must be a 1 to read from the DS1302
- You need to select the DS1302, make MOMI an output, write the control byte, make MOMI an input, write a dummy byte to SPI0DR, then read the data from the DS1302.

Here is an example in C:

```
PTS = PTS | 0x80;               /* Select DS1302 */

SPI0CR2 = SPI0CR2 | 0x08;       /* Make MOMI an output */

while ((SPI0SR & 0x20) == 0) ;  /* Wait until okay to transmit */
SPI0DR = 0x81;                  /* Select Seconds register for read */
while ((SPI0SR & 0x80) == 0) ;  /* Wait until transmission complete */
tmp = SPI0DR;                   /* Read data register to clear SPIF */

SPI0CR2 = SPI0CR2 & ~0x08;      /* Make MOMI an input */

while ((SPI0SR & 0x20) == 0) ;  /* Wait until okay to transmit */
SPI0DR = 0x00;                  /* Write junk byte to start SPI clock */
while ((SPI0SR & 0x80) == 0) ;  /* Wait until transmission complete */
seconds = SPI0DR & 0x7f;        /* read Seconds register; clear Clock Halt bit */

PTS = PTS & ~0x80;              /* Deselect slave */
```