BY DARIUSZ CABAN, PHD • SILESIAN UNIVERSITY OF TECHNOLOGY

Coding SPI software

THE SPI REQUIRES THREE WIRES FOR DATA TRANSFER PLUS A DEVICE-SELECT SIGNAL. DESIGNERS CAN IMPLEMENT PERIPHERAL COMMUNICATIONS USING PROCESSOR-BASED HARDWARE OR THE SOFTWARE ROUTINES THAT THIS ARTICLE PRESENTS.

A variety of peripheral devices in modern embedded systems, such as EEPROMs, ADCs, DACs, real-time clocks, thermal sensors, and display and communication controllers, have synchronous serial interfaces. These interfaces' main benefit is that only a few wires connect peripherals to a processor. Some cases require serial peripherals—for instance, when the system processor has a low I/O-pin count. While communicating with a device through a synchronous serial interface, data and a timing clock transmit over separate wires. The processor acts as the master, and a peripheral device acts as the slave. Only the master can initiate communications and generate the timing clock. The three main synchronous-serial-interface standards are Microwire from National Semiconductor (www.national.com), SPI (serial-peripheral-interface) from Motorola (www.motorola.com), and I^2C (inter-integrated circuit) from Philips (www.philips.com). Numerous proprietary synchronous serial interfaces exist, as well. Software in C enables a microcontroller from the Intel (www.intel.com) MCS-51 family to access SPI peripherals. This article explains how you can implement this software.

People often refer to SPI as a three-wire interface, but the interface bus comprises more than three wires. The three wires carry input data to that slave and output data from the slave and the timing clock. The developers from Motorola labeled the three wires MOSI (master out/slave in), MISO (master in/slave out, and SCK (serial clock). Multiple slaves can share these wires (**Reference 1** and **Figure 1**). The SPI slave also has a select input SS (slave select), and the master must generate a separate select signal for each slave in the system; a low-level signal selects most of the available slaves. Occasionally, a select signal also initiates a data transfer. If only one slave exists, you can sometimes permanently force its select input to an active level. The slave's data sheet specifies the maximum clock-frequency value. The manufacturers of slave devices also use equivalent labels for bus lines. MOSI is equivalent to SI (slave in) or DI (data in). MISO is equivalent to SO (slave out) or DO (data out), SCK approximates SCLK (which also stands for serial clock), and SS is approximately equivalent to CS (chip select). A high-level signal selects some serial devices.

LISTING 1 CONFIGURATION OF THE SPI PORT

```
#define uchar unsigned char

void SPI_configuration(uchar configuration)
{
  P1 |= 0xF0;              /* programming SPI pins high */
  SPCR = configuration;
}
```

LISTING 2 SENDING AND RECEIVING A BYTE

```
#define SPIF 0x80          /* SPI interrupt flag in SPSR */

uchar SPI_transfer(uchar byte)
{
  SPDR = byte;             /* byte to send */
  while(!(SPSR & SPIF));   /* wait until end of transfer */
  return(SPDR);            /* received byte */
}
```

SPI OPERATION

SPI's developers based its operation on the use of two 8-bit shift registers (**Figure 2**). While the master communicates with the selected slave, the two devices' shift registers connect in a ring, so both devices always simultaneously send and re-

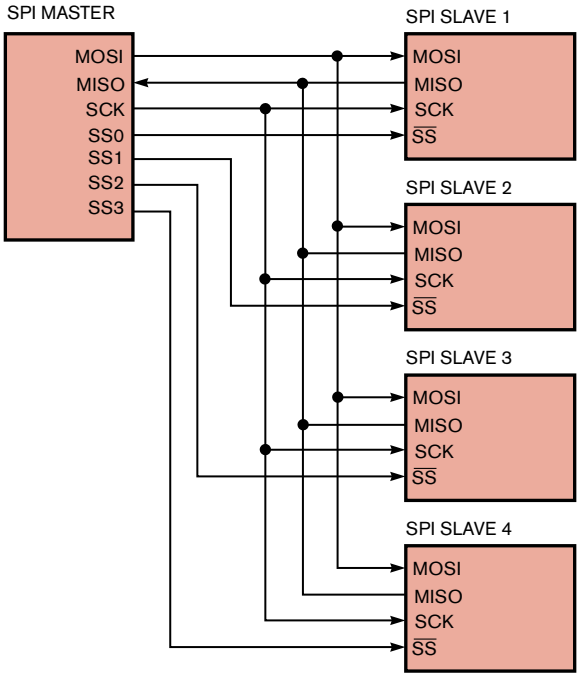

Figure 1 An embedded system comprises a few SPI peripherals under the control of one master.

ceive. If the dialogue between them requires only half-duplex communication, each device discards the bytes it received in the transmitting phase and generates dummy bytes in the receiving phase. A pair of parameters, CPOL (clock polarity) and CPHA (clock phase), defines the SPI mode. These parameters are binary digits, so there are four possible modes. CPOL selects the level of the SCK line before and after byte transfer. CPHA determines the edges of the clock on which a slave latches input-data bits and shifts out bits of output data. A master/slave pair must use the same mode to communicate. **Figure 3** presents the timing diagrams of a byte transfer in all modes.

Assume that the clock edges are numbered from one. When the CPHA equals zero, input-data bits latch onto each odd clock edge, and output-data bits shift out onto an even clock edge. The select signal initiates a byte transfer, and the first bit of output data is available after activating this signal. When a byte transfer terminates, the select line must deactivate. When CPHA equals one, input-data bits latch onto each even clock edge, and output-data bits shift out onto each odd clock edge. The first clock edge indicates the start of a byte transfer. The SS line may remain at its active level between transfers of successive bytes; a slave considers a byte transfer complete after the eighth bit latches. If there is one slave in the system, its select input may sometimes permanently remain at the active level. In 0,0 and 1,1 modes, input-data bits latch on the rising clock edges, and output-data bits shift out on the falling clock edges. The remaining modes use falling and rising clock edges.

Numerous available slave devices support both 0,0 and 1,1 modes. You access these devices using commands that often require the transfer of multiple bytes. You must select these devices before transfer of each command and deselect them after transfer of each command.

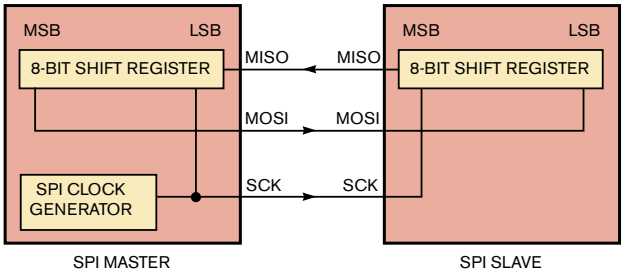

Figure 2 Each SPI device contains an 8-bit shift register. The registers of the master and selected slaves connect in a ring, allowing full-duplex communication to take place.

PORT IMPLEMENTATIONS

Motorola first included a hardware-SPI port in the 68HC11 family of microcontrollers and then extended the port to many other microcontrollers. Microcontrollers from other manufacturers, such as Atmel's (www.atmel.com) AT89S8253, also support SPI (**Reference 2**). This microcontroller is an extended 8052 device with flash program memory, which you can reprogram in a target system through SPI. Its SPI port provides master or slave operation, normal or enhanced mode, programmable-SPI mode, MSB (most-significant-bit)- or LSB (least-significant-bit)-first data transfer, four programmable SCK frequencies, an end-of-transmission interrupt flag, write-collision flag protection, a double-buffered receiver, a double-buffered transmitter in enhanced mode, and a wake-up from idle mode in slave mode.

In normal mode, three SFRs (special-function registers) control access to the port (**Figure 4**), and the microcontroller's data sheet describes those registers. **Listings 1** through **7**, written in Keil C51, illustrate the use of the port (**Reference 3**). The header file, which comes with the compiler, includes a list of the addresses of SFRs available on the AT89S8253. **Listing 1** shows the routine configuring the SPI port. If you enable the SPI port, it uses pins of the high nibble of Port 1 (P1.4 through SS/, P1.5 through MOSI, P1.6

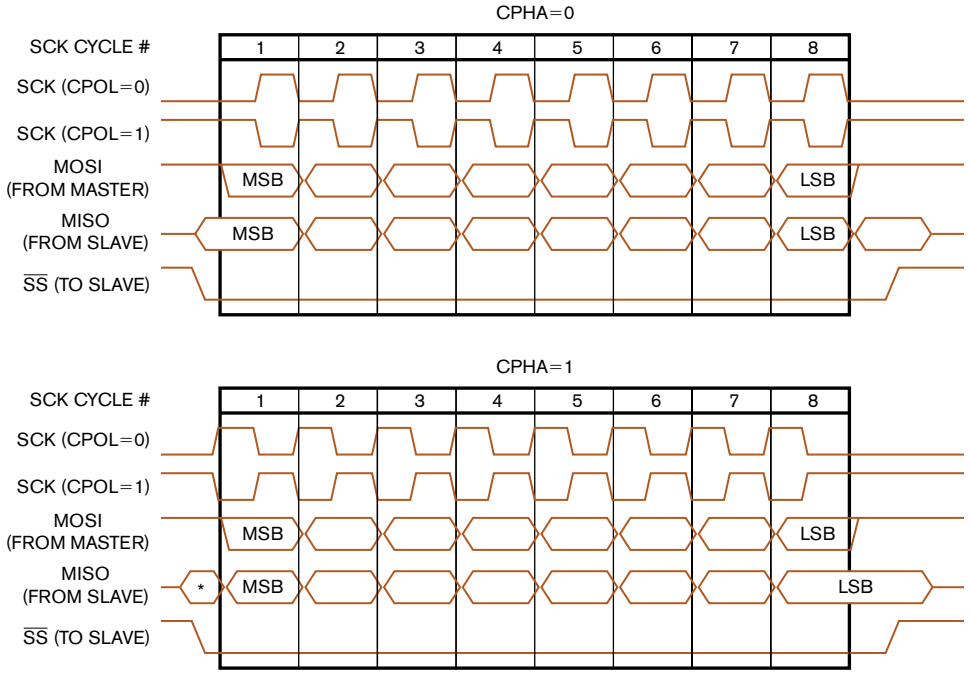

Figure 3 A master/slave pair must use the same mode to communicate. The timing diagrams of a byte transfer in all modes yield these timing profiles.

LISTING 3 BIT-BANGING SPI-TRANSFER ROUTINES

```
sbit MOSI = P1 ^ 0;     /* this declaration assigns pins of */
sbit MISO = P1 ^ 1;     /* Port 1 as SPI pins */
sbit SCK  = P1 ^ 2;

/* a byte transfer in (0,0) mode */

uchar SPI_transfer(uchar byte)
{
uchar counter;

  for(counter = 8; counter; counter--)
  {
    if (byte & 0x80)
      MOSI = 1;
    else
      MOSI = 0;
    byte <<= 1;
    SCK = 1;     /* a slave latches input data bit */
    if (MISO)
      byte |= 0x01;
    SCK = 0;     /* a slave shifts out next output data bit */
  }
  return(byte);
}

/* a byte transfer in (1,1) mode */

uchar SPI_transfer(uchar byte)
{
uchar counter;

  for(counter = 8; counter; counter--)
  {
    if (byte & 0x80)
      MOSI = 1;
    else
      MOSI = 0;
    SCK = 0;         /* a slave shifts out output data bit */
    byte <<= 1;
    if (MISO)
      byte |= 0x01;
    SCK = 1;         /* a slave latches input data bit */
  }
  return(byte);
}
```

LISTING 4 BIT'S ORDER INVERSION IN A BYTE

```
uchar inverse(uchar byte)
{
uchar mask = 1, result = 0;

  while(mask)
  {
    if (byte & 0x80)
      result |= mask;
    mask <<= 1;
    byte <<= 1;
  }
  return(result);
}
```

through MISO, and P1.7 through SCK). Atmel recommends that you set these pins high before writing to the control register; otherwise, the SPI port may not operate correctly. You must program the SPI pins in **Listing 1** if you need to reconfigure the port while the microcontroller executes its program. You can omit this operation if configuration occurs only once because a hardware reset sets the SPI pins high.

The AT89S8253 can act as an SPI master or an SPI slave, but this article considers only master operation. **Listing 2** presents a routine that sends and receives a byte through the SPI port. Writing to the SPI-data register initiates a transfer, starts the clock generator, and shifts out the output byte on the MOSI pin. Simultaneously, a byte from a slave shifts into the SPI-data register. The While loop executes until you set the SPI-interrupt flag in the SFR, which indicates the end of transfer. You clear this flag by reading the status register by setting the SPI-interrupt-flag bit and then accessing the data register.

SPCR-SPI CONTROL REGISTER (ADDRESS D5H)

SPIE	SPE	DORD	MSTR	CPOL	CPHA	SPR1	SPR0

SPSR-SPI STATUS REGISTER (ADDRESS AAH)

SPIF	WCOL	LDEN	—	—	—	DISSO	ENH

SPDR-SPI DATA REGISTER (ADDRESS 86H)

SPD7	SPD6	SPD5	SPD4	SPD3	SPD2	SPD1	SPD0

Figure 4 SFR registers control the hardware-SPI port of the AT89S8253 microcontroller in normal mode.

TABLE 1 COMMAND SET FOR THE CAT25040

Command	Command's format (a=address bit; d=data bit)			Operation
WREN	00000110	–	–	Enable write operations
WRDI	00000100	–	–	Disable write operations
RDSR	00000101	$d_7 \div d_0$[1]	–	Read status register
WRSR	00000001	$d_7 \div d_0$	–	Write status register
READ	$0000a_8011$	$a_7 \div a_0$	$d7 \div d_0$[1]	Read data from memory
WRITE	$0000a_8010$	$a_7 \div a_0$	$d7 \div d_0$[2]	Write data to memory

[1] Data bits transfer on MISO line; a single read command can read any number of bytes.
[2] A single write command can write as many as 16 bytes.

You can generate an interrupt request after the transfer completes, but this feature is more useful in slave operation. Writing to the SPI-data register during a transfer sets the WCOL (write-collision)-flag bit in the SPI-status register. This operation has no effect, and the result of reading the data register may be incorrect. Reading the status register with WCOL bit set, followed by accessing the data register, clears this flag. Using the SPI-transfer routine only to communicate with slave devices prevents collisions.

Microcontrollers without hardware support for SPI also can communicate with SPI devices, because it is feasible to perform a "bit-banging," an all-software port implementation. Any microcontroller's general-purpose-I/O pins can serve as SPI pins. Most slave devices support both 0,0 and 1,1 SPI modes; to communicate with these devices, you can use one of the equivalent SPI-transfer routines in **Listing 3**.

If a slave device supports only one mode, you must ensure that you forced the SCK line to the proper level before selecting the device. The hardware-SPI port features MSB- or LSB-first data transfer, and bit-banging routines always send MSB first. If a slave in the system requires LSB first, you can inverse the bits' order in a byte that passes to the SPI-transfer

routine and a byte that this routine returns. The routine in **Listing 4** performs this inversion. If you compare all versions of the SPI-transfer routine with respect to code-memory occupation and achievable bit rates in both SPI-port implementations, you will find that the routine that uses hardware-based SPI occupies 10 bytes of code memory, and the bit-banging routines occupy 29 bytes each if you pass in the parameters and place the local variables in the register. The AT89S8253 has one- and two-times clock options, and the machine cycles for these clocks take 12 and six oscillator periods, respectively. Maximum SCK frequency is f_{OSC}/N, where f_{OSC} is the oscillator frequency and N is 4 in the one-times mode and 2 in the two-times mode. Because it takes one clock cycle to transfer one data bit, the maximum bit rate equals f_{OSC}/N bps. A byte transfer using the bit-banging routine takes a minimum of 111 machine cycles, so the maximum bit rate is $8/(111\times t_{CYCLE})$ bps, where t_{CYCLE} is the cycle duration. For example, a classic 8051 microcontroller with a 12-MHz crystal can transmit SPI data at approximately 72 kbps.

SERIAL EEPROM

Designers often use EEPROMs as slave devices in an inexpensive approach to storing data in nonvolatile memory. Several manufacturers offer serial EEPROMs in capacities of 1 to 64 kbits or more. **Listing 5** presents software that allows access to a Catalyst Semiconductor (www.catsemi.com) CAT25040 device or equivalent (**Reference 4**). The CAT25040 provides 512 bytes of nonvolatile memory with 100-year data retention.

LISTING 5 CAT25040 READ OPERATIONS

```
#define uint  unsigned int
#define RDSR  5            /* codes of commands         */
#define READ  3

sbit EEPROM_SEL = P1 ^ 4;  /* select pin for the EEPROM */

uchar EEPROM_byte_read(uint address)
{
uchar byte;

  EEPROM_SEL = 0;
  SPI_transfer((address >> 8) ? (READ | 8) : READ);
  SPI_transfer((uchar)address);
  byte = SPI_transfer(0xFF);
  EEPROM_SEL = 1;
  return(byte);
}

void EEPROM_sequential_read(uint address, uchar
*destination,
                            uint size)
{
  if (size)
  {
    EEPROM_SEL = 0;
    SPI_transfer((address >> 8) ? (READ | 8) : READ);
    SPI_transfer((uchar)address);
    for( ; size; size--, destination++)
      *destination = SPI_transfer(0xFF);
    EEPROM_SEL = 1;
  }
}

uchar EEPROM_status_read(void)
{
uchar status;

  EEPROM_SEL = 0;
  SPI_transfer(RDSR);
  status = SPI_transfer(0xFF);
  EEPROM_SEL = 1;
  return(status);
}
```

It provides 1 million write/erase cycles and supports 0,0 and 1,1 SPI modes with a maximum SCK frequency of 10 MHz. In addition to the SPI pins, the CAT25040 has two other pins. The Hold pin enables the master to pause communication with the

LISTING 6 CAT25040 WRITE OPERATIONS

```
#define WREN  6          /* codes of commands */
#define WRSR  1
#define WRITE 2

bit EEPROM_byte_write(uint address, uchar byte)
{
  EEPROM_SEL = 0;        /* writing enable */
  SPI_transfer(WREN);
  EEPROM_SEL = 1;

  EEPROM_SEL = 0;        /* write data     */
  SPI_transfer((address >> 8) ? (WRITE | 8) : WRITE);
  SPI_transfer((uchar)address);
  SPI_transfer(byte);
  EEPROM_SEL = 1;

  return(programming_status());
}

bit EEPROM_page_write(uint address, uchar *source,
                      uchar size)
{
  if (!size || (uchar)(((uchar)address & 15) + size) > 16)
    return(0);           /* invalid number of bytes or they would
                            not occupy adjacent locations */

  EEPROM_SEL = 0;      /* writing enable */
  SPI_transfer(WREN);
  EEPROM_SEL = 1;

  EEPROM_SEL = 0;      /* write data     */
  SPI_transfer((address >> 8) ? (WRITE | 8) : WRITE);
  SPI_transfer((uchar)address);
  for( ; size; size--, source++)
    SPI_transfer(*source);
  EEPROM_SEL = 1;

  return(programming_status());
}

bit EEPROM_status_write(uchar status)
{
  EEPROM_SEL = 0;       /* writing enable */
  SPI_transfer(WREN);
  EEPROM_SEL = 1;

  EEPROM_SEL = 0;       /* write status   */
  SPI_transfer(WRSR);
    SPI_transfer(status);
    EEPROM_SEL = 1;

    return(programming_status());
  }
```

LISTING 7 PROGRAMMING-STATUS ROUTINE

```
#define RDY 1          /* READY bit in the status register
*/

bit programming_status(void)
{
uchar counter;

  for(counter = 16; counter; counter--)
  {
    delay(84);         /* about 0.5 ms, when fosc = 12 MHz
*/
    if (!(EEPROM_read_status() & RDY))
      return(1);       /* OK      */
  }
  return(0);           /* failure */
}

/* suspension of program execution for (number * 6) + 1
   machine cycles */

void delay(uchar number)
{
  while(number--);
}
```

EEPROM if another slave requires urgent servicing. The WP (write-protect) pin allows enabling and disabling writes to the memory array and the memory's status register. Enabling writing allows two or more nonvolatile bits in the status register to protect all or a portion of the memory array. In addition, you must set a write-enable latch before any write operation occurs.

MORE AT EDN.COM

Go to www.edn.com/ms4256 and click on Feedback Loop to post a comment on this article.

You access the CAT25040 using six commands (**Table 1**). The first byte is the command's code. The codes of the read and write commands contain the MSB of the location's address. You must select the memory before the transfer of each command and deselect it after the transfer. **Listing 5** presents sample routines performing read operations. After the EEPROM receives the read command's code and address, the address loads into an address counter, and the memory responds with a byte stored at the given address. The master can read the sequence of data by continuing to provide clocking. The address counter automatically increments to the next address after each byte shifts out. When the EEPROM reaches the highest address, the next address equals zero. The sequential read is a convenient way to get multibyte values from the EEPROM. You use the separate routine to read the status of the memory.

Listing 6 provides sample routines performing write operations. Before any write occurs, the master must set the write-enable latch in the EEPROM by issuing the write-enable command. Next, the master sends the write command's code, followed by the address, which loads into the address counter, and the data to write. The master can write as many as 16 bytes—a page write—by continuing to provide clocking. Compatible EEPROMs offer different page sizes. The address counter's bits constitute a page number, and the remaining bits address bytes within the page. After the EEPROM receives each byte, it increments only the address within the page. When the EEPROM reaches the highest address, the next address is zero, and if the clock continues, it may overwrite other data. To prevent this situation, the EEPROM page-write routine checks whether all bytes to write will occupy an area of consecutive addresses. If not, the routine does not issue a write command. A separate routine writes the memory's status register.

When the master deselects the EEPROM after issuing a write command, the memory enters the internal programming cycle. This cycle takes as long as 5 msec. The memory then ignores all commands except the read-status-register command. The LSB of the memory's status register indicates whether the programming cycle is in progress or complete. The programming-status routine checks this bit every 0.5 msec (**Listing 7**). The number of checks is limited; exceeding the limit indicates failure of the write operation. After the end of the programming cycle, the device is write-protected.

A range of serially accessed peripheral devices finds use in embedded systems. Connecting them to a processor requires a few wires. Such devices typically include a synchronous interface, of which the SPI is one of the most popular (**references 5** and **6**).EDN

REFERENCES

1. *M68HC11 Reference Manual, Revision 6.1*, Freescale Semiconductor Inc, 2007, www.freescale.com/files/microcontrollers/doc/ref_manual/M68HC11RM.pdf?fsrch=1.
2. "AT89S8253: 8-bit microcontroller with 12K Bytes Flash and 2K Bytes EEPROM," Atmel Corp, 2002, ftp://ftp.efo.ru/pub/atmel/AT89S8253.pdf.
3. "C51 Compiler, Optimizing C Compiler and Library Reference for Classic and Extended 8051 Microcontrollers," Keil Software Inc, 2001, www.keil.com/c51.
4. "CAT25010/20/40, 1K/2K/4K SPI Serial CMOS EEPROM," Catalyst Semiconductor Inc, www.catsemi.com/documents/25040.pdf.
5. Eck, Art, "Serial Interface for Embedded Design, Circuit Cellar, January 2000, http://i.cmpnet.com/chipcenter/circuitcellar/january00/pdf/c0100aepdf.pdf.
6. "Serial-control multiplexer expands SPI chip selects," *Maxim Engineering Journal, Volume 31*, pg 11, http://pdfserv.maxim-ic.com/en/ej/EJ31.pdf.

AUTHOR'S BIOGRAPHY

Dariusz Caban is a lecturer at the Silesian University of Technology's Institute of Informatics (Gliwice, Poland), where he has worked for seven years. In his spare time, he enjoys history and tourism.