Keywords: SPI,electricity meter,microcontroller,communication

APPLICATION NOTE 4246

How to Use Serial Peripheral Interface (SPI) on the MAXQ3180 Microcontroller

By: Ben Smith
Jun 17, 2008

Abstract: The MAXQ3180 microcontroller is a polyphase analog front-end for an electricity meter. It incorporates all functions needed for modern, multifunction electricity metering. The MAXQ3180 communicates its readings to a host microcontroller by means of a Serial Peripheral Interface (SPI™) bus. This application note describes how this interface is done, and presents sample code to assist the designer in implementing the communications mechanism.

Overview of SPI

The Serial Peripheral Interface (SPI) is an interdevice bus protocol that provides fast, synchronous, full-duplex communications between chips. One device, the *master*, drives the synchronous clock and selects which of several *slaves* is being addressed. Every SPI peripheral consists of a single shift register and control circuitry so that an addressed serial peripheral interface SPI peripheral is simultaneously transmitting and receiving.

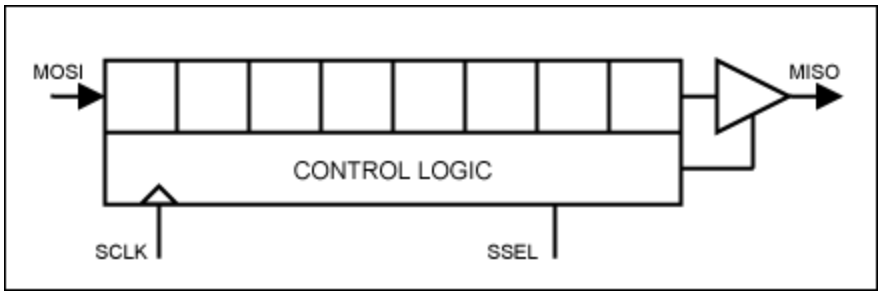

Figure 1. Illustration for a SPI slave.

There are four discrete circuits used in SPI communication:

- **SCLK**: The synchronous clock used by all devices. The master drives this clock and the slaves receive the clock. Note that SCLK can be gated and need not be driven between SPI transactions.
- **MOSI**: Master out, slave in. This is the main data line driven by the master to all slaves on the SPI bus. Only the selected slave clocks data from MOSI.
- **MISO**: Master in, slave out. This is the main data line driven by the selected slave to the master. Only the selected slave may drive this circuit. In fact, it is the only circuit in the SPI bus arrangement that a slave is *ever* permitted to drive.
- **SSEL**: This signal is unique to each slave. When active (generally low) the selected slave must

drive MISO.

For this discussion, it is critical to note that *a SPI peripheral transmits and receives at the same time*. A handy way to think of this is that the master always sends one byte and receives one byte.

Some SPI peripherals sacrifice speed in favor of simulating a half-duplex operation. This is not the case with the MAXQ3180 microcontroller, which is truly a full-duplex SPI slave.

The remainder of this application note describes how to connect and successfully use the MAXQ3180 on a SPI bus.

MAXQ3180 Communications Overview

To the master (i.e., host), the MAXQ3180 looks like a memory array that consists of both RAM and ROM. This is because the ROM firmware in the MAXQ3180 reads its operational parameters from RAM, and places its results in RAM. Consequently, configuring a MAXQ3180 is as simple as performing a block write to its RAM locations.

Some MAXQ3180 "memory" locations trigger actions within the device to calculate electricity metering results "on the fly." Writes to these locations are a "nop." The specific function and purpose of RAM and virtual ROM locations are beyond the scope of this document. The important fact here is that the microcontroller really has only two operations for SPI communication: read and write.

Every transaction in the MAXQ3180 begins with the master sending two bytes that contain the command (i.e., read or write), the address to access, and the number of bytes to access. As mentioned above, every SPI peripheral returns a byte for every byte received. The MAXQ3180, therefore, returns 0xC1 after receiving the first command byte, and 0xC2 after the second command byte. The protocol is illustrated below in **Figure 2**.

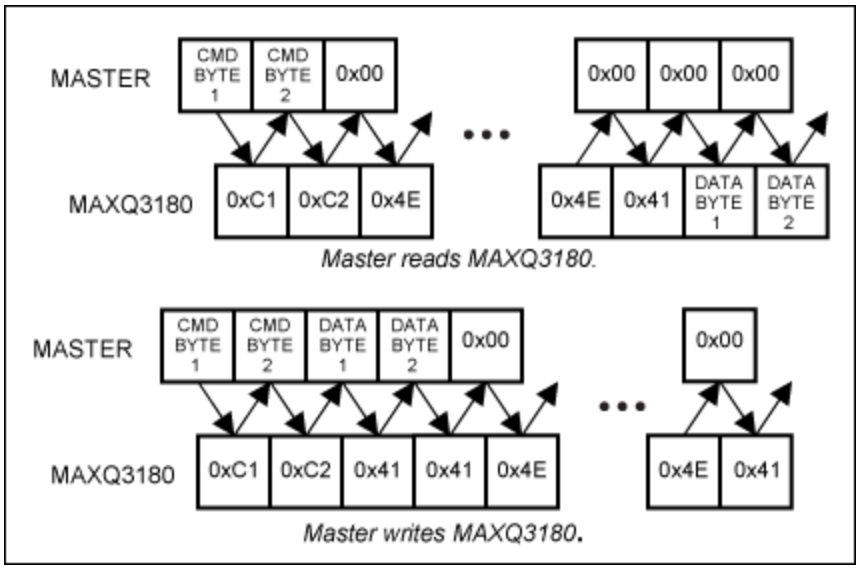

Figure 2. Master reads and writes data to the MAXQ3180.

If the master is reading one or more bytes, it must send dummy bytes. Remember that the master cannot receive *anything* from the slave unless it sends something: send a byte to get a byte. But after receiving a command, the MAXQ3180 may have to calculate the result and thus may not have it ready when the master sends the dummy byte. For this reason, the MAXQ3180 *always* sends zero or more

bytes of a NAK character (0x4E or ASCII 'N') followed by an ACK character (0x41, or ASCII 'A') before sending the data.

If the master is writing one or more bytes, it sends the data to be written *immediately* after sending the command. The MAXQ3180 returns ACK (0x41) for each data byte. It then returns NAK (0x4E) until the write cycle is complete, after which it returns a final ACK.

Note that immediately after the final ACK, the MAXQ3180 is ready to begin the next transaction; it does not need to wait for any other event. It is not even necessary to toggle SSEL to begin the next transaction. The MAXQ3180 knows that the first transaction is over and is ready for the next.

If, for whatever reason, it is necessary to reset the communications between the master and the MAXQ3180 (for example, if communication becomes unsynchronized) the master need only wait 200ms before restarting communication from the first command byte. A 200ms delay informs the MAXQ3180 that the master is abandoning the previous transaction.

Command Bytes

The command bytes tell the MAXQ3180:

1. Whether the requested transaction is a READ or WRITE
2. The length of the transaction
3. The address in RAM to modify (or the virtual ROM address to read)

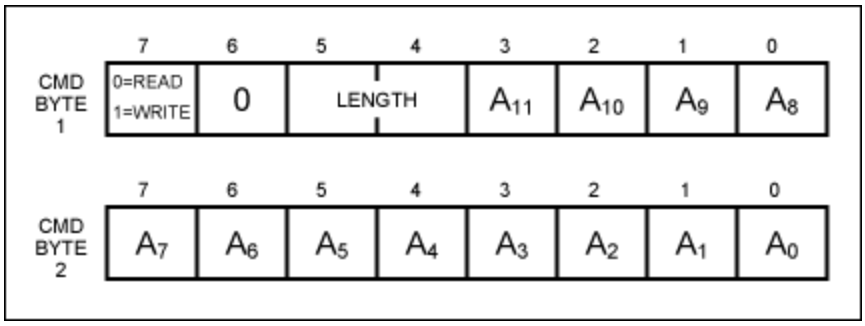

Figure 3. Structure of the command bytes.

The first command byte (**Figure 3**) tells the MAXQ3180 whether the transaction being requested is a READ or a WRITE, and the length of the transaction. The command byte uses the following schedule:

Length Code	Data Length
0b00	1 byte
0b01	2 bytes
0b10	4 bytes
0b11	8 bytes

The remainder of command byte 1 and all of command byte 2 provide the address of the byte in RAM (or the identity of the virtual ROM function) to be accessed.

Host Software Design

Although the MAXQ3180 contains a hardware SPI controller, individual message bytes are still processed in a software routine contained in the ROM firmware. For this reason, a delay is necessary

between successive bytes. In current versions of the MAXQ3180, this delay must be no less than 100µs for reliable operation. See **Figures 4** and **5**.

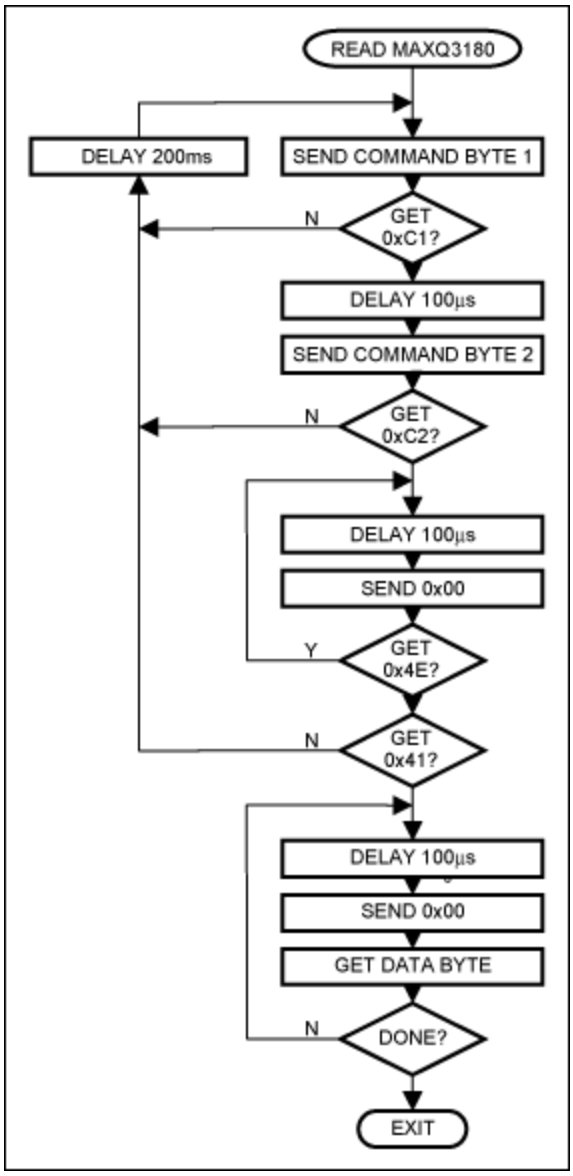

Figure 4. Flowchart for reading from the MAXQ3180.

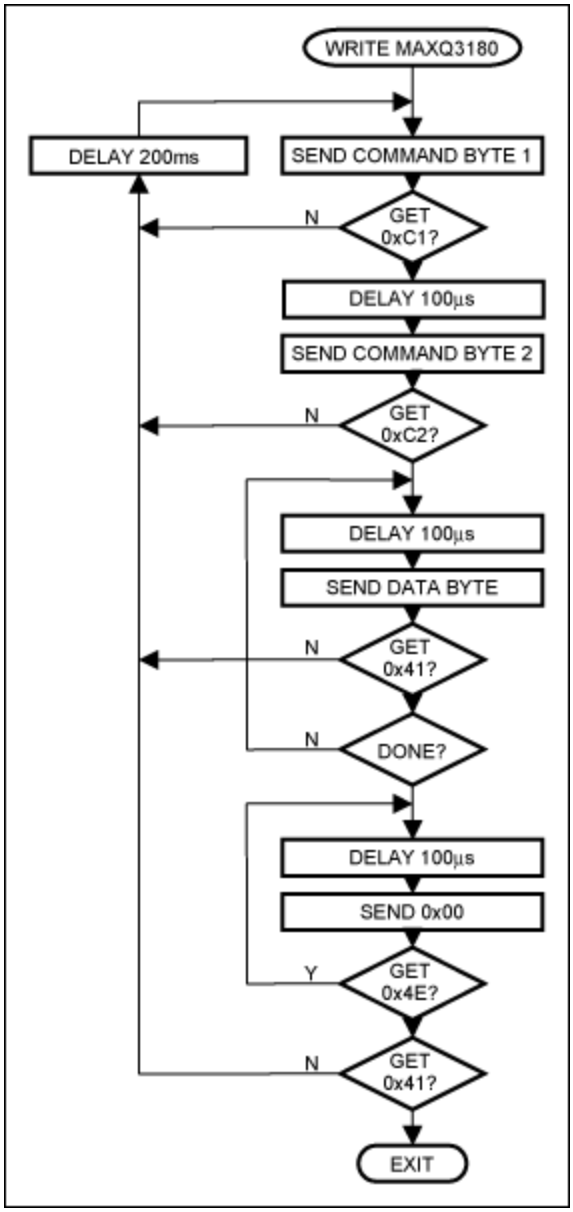

Figure 5. Flowchart for writing the MAXQ3180.

Code Listings

Code is provided for interfacing a MAXQ2000 microcontroller with a built-in SPI master to the MAXQ3180. Users of other microcontrollers will need to provide their own SPI primitives and possibly modify the high-level subroutines as well.

In the listings below, the **`dly_us`** subroutine causes the program thread to cease execution for the given number of microseconds. The **`SPI_TIMEOUT`** constant is defined to provide something longer than a character timeout.

In the high-level subroutines, an ENUM is used to select the register by name. It indexes the **`register_lookup_table`** array that, among other things, contains the register length for each of the MAXQ3180 registers. See **Figures 6**, **7**, and **8**.

```
unsigned char Send_SPI(unsigned char x)
{
  unsigned char y = 0;
  int z;
  int error = 0;
  SPICN = 3; /* MSTSM, SPIEN */
  z = 0; while ((SPICN_bit.STBY) && (++z < SPI_TIMEOUT));
  if (z == SPI_TIMEOUT) error = 1;
  SPICN_bit.SPIC = 0; /* Clear transfer complete flag */
  SPIB = x;
  z = 0; while ((!SPICN_bit.SPIC) && (++z < SPI_TIMEOUT));
  if (z == SPI_TIMEOUT) error = 1;
  y = SPIB;
  SPICN_bit.SPIC = 0;
  dly_us(100);
  if (error) return 0;
  return y;
}
```

Figure 6. Code for the Send_SPI primitive.

```
long Read_AFE(enum METER_REGISTER_RECORD reg, uint16 reg_addr)
{
  extern unsigned char record[8];
  unsigned long x = 0;
  unsigned char i, regadd, command_code = 0;
  for(i=0; i<8; i++) record[i] = 0;
  switch(register_lookup_table[reg].register_length)
  {
  case 2: command_code |= 0x10; break;
  case 4: command_code |= 0x20; break;
  case 8: command_code |= 0x30; break;
  }
  command_code |= reg_addr >> 8;
  regadd = reg_addr & 0xff;

  /* Disable SPI to reset it */
  SPICN_bit.SPIEN = 0;
  for(x=0; x<300; x++);
  SPICN_bit.SPIEN = 1;

  SPI_SELECT_0;
  i = 0;
  while((Send_SPI(command_code)!= 0xC1)&&(++i < SPI_COMMAND_RETRIES))
    spi_comm_timeout();

  x = 0xffffffff;
  if (i == SPI_COMMAND_RETRIES) goto spierror;
  Send_SPI(regadd);
  i = 0; while((Send_SPI(0) != 'A') && (++i < SPI_RETRIES));
  if (i == SPI_RETRIES) goto spierror;
  x = 0;
  for(i=0; i<register_lookup_table[reg].register_length; i++)
  {
    record[i] = Send_SPI(0);
    x |= ((uint32)record[i]) << (i * 8);
  }

spierror:
  SPI_DESELECT_0;
  return (int32)x;
}
```

Figure 7. Code for the ReadAFE (SPI_Read) subroutine.

```
void Write_AFE(enum METER_REGISTER_RECORD reg, uint16 reg_addr, uint32 data)
{
  uint8 i, regadd, command_code = 0x80;
  int x;
  switch(register_lookup_table[reg].register_length)
  {
  case 2: command_code |= 0x10; break;
  case 4: command_code |= 0x20; break;
  case 8: command_code |= 0x30; break;
  }
  command_code |= reg_addr >> 8;
  regadd = reg_addr & 0xff;

  /* Disable SPI hardware to reset it */
  SPICN_bit.SPIEN = 0;
  for(x=0; x<300; x++);
  SPICN_bit.SPIEN = 1;

  SPI_SELECT_0;
  i = 0;
  while((Send_SPI(command_code)!=0xC1)&&(++i < SPI_COMMAND_RETRIES))
    spi_comm_timeout();
  if (i == SPI_COMMAND_RETRIES) goto spierror;
  Send_SPI(regadd);
  for(i=0; i<register_lookup_table[reg].register_length; i++)
    Send_SPI((data >> (i * 8)) & 0xff);
  i = 0; while((Send_SPI(0) != 'A') && (++i < SPI_RETRIES));

spierror:
  SPI_DESELECT_0;
}
```

Figure 8. Code for a `Write_AFE (SPI_Write)` subroutine.

Related Parts		
MAXQ2000	Low-Power LCD Microcontroller	Free Samples
MAXQ3180	Low-Power, Multifunction, Polyphase AFE	Free Samples

More Information
For Technical Support: http://www.maximintegrated.com/support
For Samples: http://www.maximintegrated.com/samples
Other Questions and Comments: http://www.maximintegrated.com/contact

Application Note 4246: http://www.maximintegrated.com/an4246
APPLICATION NOTE 4246, AN4246, AN 4246, APP4246, Appnote4246, Appnote 4246