Keywords: Video, On-Screen Display, Security Cameras, DVRs, Switchers, Composite Video, Video Switch Matrix, CCTV Cameras, OSD, Video Monitors, Video Recorders, PTZ Cameras, SPI Interface, Software Routines, MAX7456, Monochrome OSD, Code for MAX7456

Mar 31, 2008

APPLICATION NOTE 4184

C-Code Software Routines for Using the SPI Interface on the MAX7456 On-Screen Display

Abstract: The MAX7456 on-screen display (OSD) generator has an SPI™-compatible control interface. This application note describes the operation of the SPI interface. The article also includes C-code that a microcontroller can use to control the part through a bit-banged SPI interface.

The MAX7456 Serial Interface

The MAX7456 single-channel, monochrome on-screen display (OSD) generator is preloaded with 256 characters and pictographs, and can be reprogrammed in-circuit using the SPI port. The SPI-compatible serial interface programs the operating modes, the display memory, and the character memory. Read capability permits both write verification and reading of the Status (STAT), Display Memory Data Out (DMDO), and Character Memory Data Out (CMDO) registers. For detailed information on the MAX7456 registers and memory organization, refer to the product data sheet and to application note 4117, "Generating Custom Characters and Graphics by Using the MAX7456's Memory and EV Kit File Formats."

The MAX7456 supports interface clocks (SCLK) up to 10MHz. **Figure 1** illustrates writing data and **Figure 2** illustrates reading data from the device.

To write to a register, bring active-low CS low to enable the serial interface. Data is clocked in at SDIN on the rising edge of SCLK. When active-low CS transitions high, data is latched into the input register. If active-low CS goes high in the middle of a transmission, the sequence is aborted (i.e., data is not written to the registers). After active-low CS is brought low, the device waits for the first byte to be clocked into SDIN before it can identify the type of data transfer being executed.

To read a register, bring active-low CS low as described above. The address is clocked in at SDIN on the rising edge of SCLK, as described above. The data is then clocked out at SDOUT on the falling edge of SCLK.

The SPI commands are 16 bits long; the 8 most significant bits (MSBs) represent the register address and the 8 least significant bits (LSBs) represent the data (Figures 1 and 2). There are two exceptions to this arrangement:

1. Autoincrement write mode, used for display memory access, is a single 8-bit operation (**Figure 3**). The start address must be set before the data is written. When performing the autoincrement write for the display memory, the 8-bit address is internally generated; only 8-bit data is required at the serial interface, as shown in Figure 3.
2. Reading character data from the Display Memory, when in 16-bit Operation Mode, is a 24-bit operation (8-bit address + 16-bit data).

When performing a read operation, only an 8-bit address is required, as shown in Figure 2.

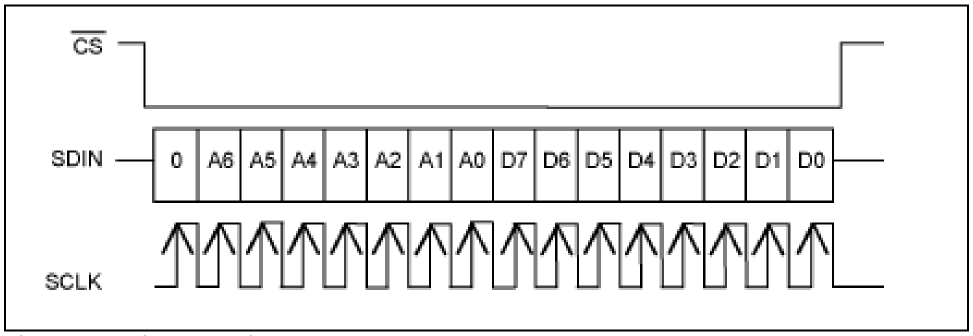

Figure 1. Write operation.

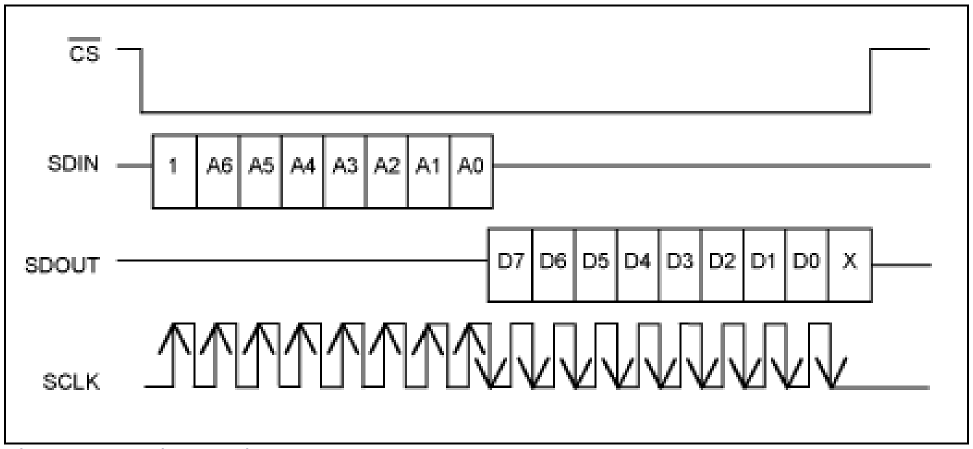

Figure 2. Read operation.

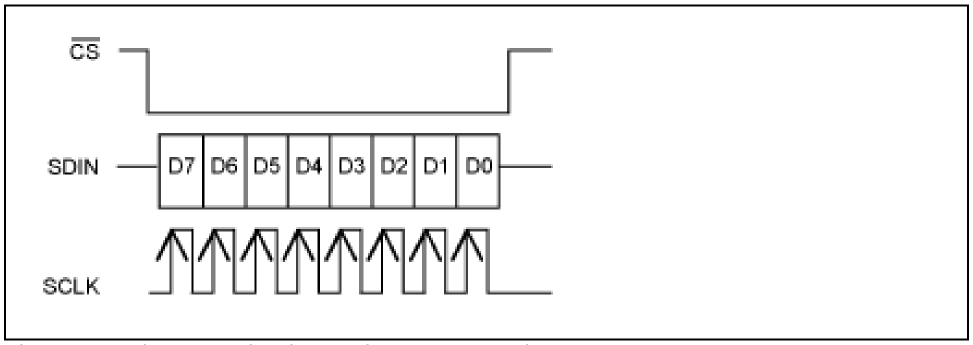

Figure 3. Write operation in autoincrement mode.

C-Code Routines

The C-code described below has been compiled for the MAXQ2000 microcontroller, which is used on the MAX7456 evaluation (EV) kit. The complete set of software routines is available in this application note. The routines are self-documenting, so little additional description is provided. The C-Code below is also available in the following files: spi.c and MAX7456.h

The code uses the standard nomenclature for the SPI lines. The MAXQ2000 processor is the SPI master and the MAX7456 is the SPI slave.

CS is the same as is used in the MAX7456 data sheet.
SDIN is referred to as MOSI (master out slave in).
SDOUT is referred to as MOSI (master in slave out).
SCLK is referred to as CK.

The prefix SPI_ is used on all lines.

Data Structures

The data structure shown below is used to access data directly or bit by bit. This is used to access the pins for the SPI port individually. (C++ and some newer C compilers support the bit-field union/struct syntax.)

```
/* Port 5 Output Register */
__no_init volatile __io union
{
  unsigned char PO5;
  struct
  {
    unsigned char bit0          : 1;
    unsigned char bit1          : 1;
    unsigned char bit2          : 1;
    unsigned char bit3          : 1;
    unsigned char bit4          : 1;
    unsigned char bit5          : 1;
    unsigned char bit6          : 1;
    unsigned char bit7          : 1;
  } PO5_bit;
}
```

This code assigns a single byte to PO5, which is the address of the microcontroller's output port. It then assigns another byte to the same memory address that can be accessed bit by bit.

Therefore, the port can be addressed directly by using commands like:

PO5 = 0x10;

Or bit by bit by using commands like:

PO5_bit.bit4 = 1;

This structure can be customised if the code is used on different processors.

If using an older C compiler which does not support the bitfield width specifier, the bitwise boolean operators can be used to set and clear bits:

```
/* Portable bit-set and bit-clear macros. */
#define BIT_SET(sfr,bitmask) sfr |= (bitmask)
#define BIT_CLR(sfr,bitmask) sfr &=~ (bitmask)
#define BIT0 0x01
#define BIT1 0x02
#define BIT2 0x04
#define BIT3 0x08
#define BIT4 0x10
#define BIT5 0x20
#define BIT6 0x40
#define BIT7 0x80
example: BIT_SET(PO5,BIT0); BIT_CLR(PO5,BIT6);
```

Macros

There is a simple tip to make the routines more mobile: use macros to define the controller pin assignments, as shown below.

```
#define SPI_CS         PO5_bit.bit4                            // PO5_bit.bit4 = active-low CS—chip select
#define SPI_MOSI       PO5_bit.bit5                            // PO5_bit.bit5 = MOSI—master out slave in,
                                                               // data to MAX7456
#define SPI_MISO       PI5_bit.bit7                            // PO5_bit.bit7 = MISO—master in slave out,
                                                               // data from MAX7456
#define SPI_CK         PO5_bit.bit6                            // PO5_bit.bit6 = SCK - SPI clock
```

Using these macros and the data structure above, one can set and reset each pin in an IO port individually with commands such as:

SPI_CS = 1;

Changing the macros will move the pins around. This is useful if the code is to be used in different designs that assign different pins for the SPI port, or if the pins need to be reassigned for better PCB routing.

Code for a Single-Byte Write

The code for a single-byte write operation (Figure 1) is shown below. If one can guarantee that the active-low CS and CK lines are at the correct state on entry, the first two instructions can be removed.

The routine first sends the address followed by the data. Two loops are used for this. It is possible to simplify the routine by using a single loop and a 16-bit data store. Rotating a 16-bit "int" takes longer than rotating an 8-bit "char" on the MAXQ2000 microcontroller, so a trade-off has been made.

```
/**************************************************************************************
 * spiWriteReg
 *
 * Writes to an 8-bit register with the SPI port
 **************************************************************************************/

void spiWriteReg(const unsigned char regAddr, const unsigned char regData)
{

  unsigned char SPICount;                                       // Counter used to clock out the data

  unsigned char SPIData;                                        // Define a data structure for the SPI data

  SPI_CS = 1;                                                   // Make sure we start with active-low CS high
  SPI_CK = 0;                                                   // and CK low

  SPIData = regAddr;                                            // Preload the data to be sent with Address
  SPI_CS = 0;                                                    // Set active-low CS low to start the SPI cycle
                                                                 // Although SPIData could be implemented as an "int",
                                                                 // resulting in one
                                                                 // loop, the routines run faster when two loops
                                                                 // are implemented with
                                                                 // SPIData implemented as two "char"s.

  for (SPICount = 0; SPICount < 8; SPICount++)                   // Prepare to clock out the Address byte
  {
    if (SPIData & 0x80)                                          // Check for a 1
      SPI_MOSI = 1;                                              // and set the MOSI line appropriately
    else
      SPI_MOSI = 0;
    SPI_CK = 1;                                                  // Toggle the clock line
    SPI_CK = 0;
    SPIData <<= 1;                                               // Rotate to get the next bit
  }                                                              // and loop back to send the next bit

                                                                 // Repeat for the Data byte
  SPIData = regData;                                             // Preload the data to be sent with Data
  for (SPICount = 0; SPICount < 8; SPICount++)
  {
    if (SPIData & 0x80)
      SPI_MOSI = 1;
    else
      SPI_MOSI = 0;
    SPI_CK = 1;
    SPI_CK = 0;
    SPIData <<= 1;
  }
  SPI_CS = 1;
  SPI_MOSI = 0;
}
```

Code for a Byte-Read Operation

The code for a byte-read operation (Figure 2) is shown below. It is similar to the routine above. The address is first sent, and the data is read back by toggling the clock and then reading in the data from the MISO line.

```
/**************************************************************************************
 * spiReadReg
 *
 * Reads an 8-bit register with the SPI port.
 * Data is returned.
 **************************************************************************************/

unsigned char spiReadReg (const unsigned char regAddr)
{

  unsigned char SPICount;                                        // Counter used to clock out the data

  unsigned char SPIData;

  SPI_CS = 1;                                                    // Make sure we start with active-low CS high
  SPI_CK = 0;                                                    // and CK low
  SPIData = regAddr;                                             // Preload the data to be sent with Address and Data

  SPI_CS = 0;                                                    // Set active-low CS low to start the SPI cycle
  for (SPICount = 0; SPICount < 8; SPICount++)                   // Prepare to clock out the Address and Data
  {
    if (SPIData & 0x80)
      SPI_MOSI = 1;
    else
      SPI_MOSI = 0;
    SPI_CK = 1;
    SPI_CK = 0;
    SPIData <<= 1;
  }                                                              // and loop back to send the next bit
  SPI_MOSI = 0;                                                  // Reset the MOSI data line

  SPIData = 0;
  for (SPICount = 0; SPICount < 8; SPICount++)                   // Prepare to clock in the data to be read
  {
    SPIData <<=1;                                               // Rotate the data
    SPI_CK = 1;                                                 // Raise the clock to clock the data out of the MAX7456
    SPIData += SPI_MISO;                                        // Read the data bit
    SPI_CK = 0;                                                 // Drop the clock ready for the next bit
  }                                                             // and loop back
  SPI_CS = 1;                                                   // Raise CS

  return ((unsigned char)SPIData);                              // Finally return the read data
}
```

Code for a Byte-Write Operation Using Autoincrement

The code for a byte-write operation using the autoincrement function (Figure 3) is shown below. It, too, is similar to the single-byte write routine above. The address is first sent, and the data is read back by toggling the clock and then reading in the data from the MISO line.

```
/*******************************************************************************
 * spiWriteRegAutoIncr
 *
 * Writes to an 8-bit register with the SPI port using the MAX7456's autoincrement mode
 *******************************************************************************/

void spiWriteRegAutoIncr(const unsigned char regData)
{
  unsigned char SPICount;                                       // Counter used to clock out the data

  unsigned char SPIData;                                        // Define a data structure for the SPI data.

  SPI_CS = 1;                                                   // Make sure we start with active-low CS high
  SPI_CK = 0;                                                   // and CK low
  SPIData = regData;                                            // Preload the data to be sent with Address and Data

  SPI_CS = 0;                                                   // Set active-low CS low to start the SPI cycle
  for (SPICount = 0; SPICount < 8; SPICount++)                  // Prepare to clock out the Address and Data
  {
    if (SPIData & 0x80)
      SPI_MOSI = 1;
    else
      SPI_MOSI = 0;
    SPI_CK = 1;
    SPI_CK = 0;
    SPIData <<= 1;
  }                                                             // and loop back to send the next bit
  SPI_MOSI = 0;                                                 // Reset the MOSI data line
}
```

Code to Write to the Display Memory Using Autoincrement

The following routine uses the autoincrement function to write to the display memory. The code uses a global variable array called "data". It is defined below:

```
extern volatile unsigned char data[DATA_BUF_LENGTH];

DATA_BUF_LENGTH = 968
```

When the routine is called, data[] contains the display memory to be written as below:

```
data[0] = ignored (contains a command byte used by the EV kit GUI software)
data[1] = character byte 1
data[2] = attribute byte 1
data[3] = character byte 2
data[4] = attribute byte 2
etc.
```

Autoincrement mode is terminated by writing 0xFF. Hence, character 0xFF cannot be written to the display in this mode. If this is required, a single-byte write can be used.

```
/*******************************************************************************
 * spiWriteCM
 *
 * Writes to the Display Memory (960 bytes) from "data" extern.
 * 960 = 16 rows × 30 columns × 2 planes {char vs. attr} screen-position-indexed memory
 ***************************************************************************************/

void spiWriteCM()                                                 // On entry: global data[1..960]
                                                                  // contains char+attr bytes
                                                                  // (optionally terminated by 0xFF data)
                                                                  // First, write data[1,3,5,...] Character plane;
                                                                  // MAX7456 WriteReg(0x05,0x41)
                                                                  // "Character Memory Address High";
                                                                  // 0x02:Attribute bytes;
                                                                  // 0x01:character memory address msb
{
   volatile unsigned int Index = 0x0001;                          // Index for lookup into
                                                                  // data[1..960]
   spiWriteReg(DM_ADDRH_WRITE,0x00);                              // initialise the Display Memory high-byte
   spiWriteReg(DM_ADDRL_WRITE,0x00);                              // and the low-byte
   spiWriteReg(DM_MODE_WRITE ,0x41);                              // MAX7456 WriteReg(0x04,0x41) "Display Memory Mode";
                                                                  // 0x40:Perform 8-bit operation; 0x01:AutoIncrement


   Do                                                             // Loop to write the character data
   {
      if (data[Index] == 0xFF) {                                  // Check for the break character
           break; }                                               // and finish if found
      spiWriteRegAutoIncr(data[Index]);                           // Write the character
      Index += 2;                                                 // Increment the index to the next character,
                                                                  // skipping over the attribute
   } while(Index < 0x03C1);                                       // 0x03C1 = 961
                                                                  // and loop back to send the next character

   spiWriteRegAutoIncr(0xFF);                                     // Write the "escape character" to end AutoIncrement
                                                                  // mode

   spiWriteReg(DM_ADDRH_WRITE,0x02);                              // Second, write data[2,4,6,...]
                                                                  // Attribute plane; MAX7456
                                                                  // WriteReg(0x05,0x41)
                                                                  // "Character Memory Address High";
                                                                  // 0x02:Attribute bytes; 0x01:character memory address
                                                                  // msb

   spiWriteReg(DM_ADDRL_WRITE,0x00);
   spiWriteReg(DM_MODE_WRITE,0x41);                               // MAX7456 WriteReg(0x04,0x41) "Character Memory
                                                                  // Mode"; 0x40:Perform 8-bit operation; 0x01:Auto-
                                                                  // Increment

   Index = 0x0002;
   do
   {
      if (data[Index] == 0xFF)
         break;
      spiWriteRegAutoIncr(data[Index]);
      Index += 2;
   } while(Index < 0x03C1);
   spiWriteRegAutoIncr(0xFF);
}
```

Code to Write to the Character Memory

The following routine writes a single character to the character memory. Each character is 12 pixels by 18 lines, totalling 216 pixels. Since each byte defines four pixels, 54 bytes are required to define each character. The data for the character is held in data[] on entry. (This is similar to the above routine for writing to the display memory.)

Writing to the character memory is worth some extra explanation. The memory is nonvolatile and, therefore, writing to it takes around 12ms and is performed by the MAX7456 itself. Only whole characters of 54 bytes can be written to the character memory.

The device contains a 54-byte shadow memory. This memory is first filled with the character data to be written. The device is then triggered to write this data to the NVM character memory.

There are several registers used to write to the character memory:

1. Character Memory Mode = 0x08. Write 0xA0 to this register to trigger the device to write the shadow memory to the NVM character memory.
2. Character Memory Address High = 0x09. This contains the address of the character to be written.
3. Character Memory Address Low = 0x0A.

4. Character Memory Data In = 0x0B
5. Status = 0xA0. Read from this to determine when the character memory is available for writing.

On entry, data[1] contains the address of the character to be written, data[2...54] contains the data for the character.

To write a character to the NVM character memory, first write the address of the character. Each byte is then written to the shadow memory. There is no autoincrement mode for writing to shadow memory, so the address within the shadow memory must be written each time. The shadow memory can then be written to the NVM character memory by writing 0xA0 to the Character Memory Mode register. The device will then set the Status register bit 5 high to indicate that the character memory is not available for writing. Once completed, the device will reset this bit low. No attempt should be made to write to the shadow memory while it is being transferred to the character memory.

To avoid causing objectionable display flicker, the routine disables the OSD before writing to the character memory.

```
/*************************************************************************************
 * spiWriteFM
 *
 * Writes to the Character Memory (54 bytes) from "data" extern
*************************************************************************************/
void spiWriteFM()
{
     unsigned char Index;

     spiWriteReg(VIDEO_MODE_0_WRITE,spiReadReg
                 (VIDEO_MODE_0_READ) & 0xF7);                 // Clear bit 0x08 to DISABLE the OSD display
     spiWriteReg(FM_ADDRH_WRITE,data[1]);                     // Write the address of the character to be written
                                                              // MAX7456 glyph tile definition
                                                              // length = 0x36 = 54 bytes
                                                              // MAX7456 64-byte Shadow RAM accessed
                                                              // through  FM_DATA_.. FM_ADDR.. contains a single
                                                              // character/glyph-tile shape
     for(Index = 0x00; Index < 0x36; Index++)
     {
          spiWriteReg(FM_ADDRL_WRITE,Index);                  // Write the address within the shadow RAM
          spiWriteReg(FM_DATA_IN_WRITE,data[Index + 2]);      // Write the data to the shadow RAM
     }

     spiWriteReg(FM_MODE_WRITE, 0xA0);                        // MAX7456 "Font Memory Mode" write 0xA0 triggers
                                                              // copy from 64-byte Shadow RAM to NV array.

     while ((spiReadReg(STATUS_READ) & 0x20) != 0x00);        // Wait while NV Memory status is BUSY
                                                              // MAX7456 0xA0 status bit 0x20: NV Memory Status
                                                              // Busy/~Ready
}
```

Header File for the MAX7456

The following listing is a header file for the MAX7456. This code defines the register map for the device.

```
/*************************************************************************************
 * spiWriteRegAutoIncr
 *
 * Writes to an 8-bit register with the SPI port by using the MAX7456's autoincrement mode
 *************************************************************************************/
                                                              // MAX7456 VIDEO_MODE_0 register
#define VIDEO_MODE_0_WRITE              0x00
#define VIDEO_MODE_0_READ               0x80
#define VIDEO_MODE_0_40_PAL             0x40
#define VIDEO_MODE_0_20_NoAutoSync      0x20
#define VIDEO_MODE_0_10_SyncInt         0x10
#define VIDEO_MODE_0_08_EnOSD           0x08
#define VIDEO_MODE_0_04_UpdateVsync     0x04
#define VIDEO_MODE_0_02_Reset           0x02
#define VIDEO_MODE_0_01_EnVideo         0x01
                                                              // VIDEO MODE 0 bitmap
#define NTSC                            0x00
#define PAL                             0x40
#define AUTO_SYNC                       0x00
#define EXT_SYNC                        0x20
#define INT_SYNC                        0x30
#define OSD_EN                          0x08
#define VERT_SYNC_IMM                   0x00
```

```
#define VERT_SYNC_VSYNC                 0x04
#define SW_RESET                        0x02
#define BUF_EN                          0x00
#define BUF_DI                          0x01

                                                                // MAX7456 VIDEO_MODE_1 register
#define VIDEO_MODE_1_WRITE              0x01
#define VIDEO_MODE_1_READ               0x81

                                                                // MAX7456 DM_MODE register
#define DM_MODE_WRITE                   0x04
#define DM_MODE_READ                    0x84

                                                                // MAX7456 DM_ADDRH register
#define DM_ADDRH_WRITE                  0x05
#define DM_ADDRH_READ                   0x85

                                                                // MAX7456 DM_ADDRL register
#define DM_ADDRL_WRITE                  0x06
#define DM_ADDRL_READ                   0x87

                                                                // MAX7456 DM_CODE_IN register
#define DM_CODE_IN_WRITE                0x07
#define DM_CODE_IN_READ                 0x87

                                                                // MAX7456 DM_CODE_OUT register
#define DM_CODE_OUT_READ                0xB0

                                                                // MAX7456 FM_MODE register
#define FM_MODE_WRITE                   0x08
#define FM_MODE_READ                    0x88

                                                                // MAX7456 FM_ADDRH register
#define FM_ADDRH_WRITE                  0x09
#define FM_ADDRH_READ                   0x89

                                                                // MAX7456 FM_ADDRL register
#define FM_ADDRL_WRITE                  0x0A
#define FM_ADDRL_READ                   0x8A

                                                                // MAX7456 FM_DATA_IN register
#define FM_DATA_IN_WRITE                0x0B
#define FM_DATA_IN_READ                 0x8B

                                                                // MAX7456 FM_DATA_OUT register
#define FM_DATA_OUT_READ                0xC0

                                                                // MAX7456 STATUS register
#define STATUS_READ                     0xA0
#define STATUS_40_RESET_BUSY            0x40
#define STATUS_20_NVRAM_BUSY            0x20
#define STATUS_04_LOSS_OF_SYNC          0x04
#define STATUS_02_PAL_DETECTED          0x02
#define STATUS_01_NTSC_DETECTED         0x01

                                                                // MAX7456 requires clearing OSD Black Level
                                                                // register bit 0x10 after reset
#define OSDBL_WR                        0x6C
#define OSDBL_RD                        0xEC
#define OSDBL_10_DisableAutoBlackLevel  0x10
```

Conclusion and Performance

The MAX7456 EV kit uses the MAXQ2000 microcontroller that runs at 20MHz clock. This microcontroller contains an internal hardware SPI controller. The MAX7456's SPI port can, therefore, run at full speed. The software SPI routines above do perform slower than the hardware controller. However, the routines have been optimized for portability if a customer application lacks a hardware SPI port.

SPI is a trademark of Motorola, Inc.

Related Parts

MAX7456:	QuickView -- Full (PDF) Data Sheet -- Free Samples
MAX7456EVKIT:	QuickView -- Full (PDF) Data Sheet