CHAPTER 16

SPI Bus

The Serial Peripheral Interface bus, known affectionately as *spy*, is a synchronous serial interface that was named by Motorola.[18] The SPI protocol operates in full-duplex mode, allowing it to send and receive data simultaneously. Generally speaking, SPI has a speed advantage over the I^2C protocol but requires more connections.

SPI Basics

Devices on the SPI bus communicate on a master/slave basis. Multiple slaves coexist on a given SPI bus, with each slave being selected for communication by a slave select signal (also known as chip select). Figure 16-1 shows the Raspberry Pi as the master communicating with one slave device. Additional slaves would be connected as shown with the exception that a different slave select signal would be used.

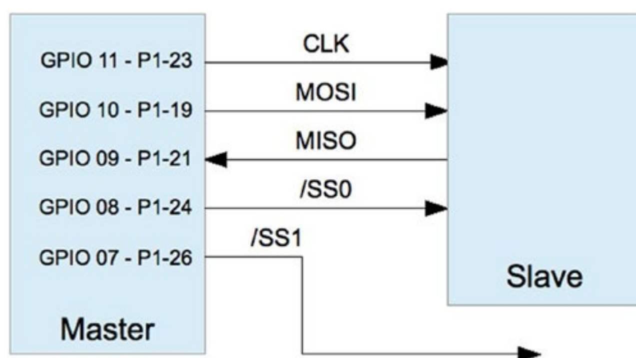

Figure 16-1. *SPI interface*

© Warren Gay 2018
W. Gay, *Advanced Raspberry Pi*, https://doi.org/10.1007/978-1-4842-3948-3_16

Data is transmitted from the master to the slave by using the MOSI line (master out, slave in). As each bit is being sent out by the master, the slave simultaneously sends data on the MISO line (master in, slave out). Bits are shifted out of the master and into the slave, while bits are shifted out of the slave and into the master. Both transfers occur to the beat of the system clock (CLK).

Many SPI devices support only 8-bit transfers, while others are more flexible. The SPI bus is a de facto standard, meaning that there is no standard for data transfer width and SPI mode.[18] The SPI controller can also be configured to transmit the most significant or the least significant bit first. All of this flexibility can result in confusion.

SPI Mode

SPI operates in one of four possible clock signaling modes, based on two parameters:

Parameter	Description
CPOL	Clock polarity
CPHA	Clock phase

Each parameter has two possibilities, resulting in four possible SPI modes of operation. Table 16-1 lists all four available modes. Note that a given mode is often referred to by using a pair of numbers like *1,0* or simply as mode *2* (for the same mode, as shown in the table). Both references types are shown in the Mode column.

Table 16-1. SPI Modes

CPOL	CPHA	Mode		Description
0	0	0,0	0	Noninverted clock, sampled on rising edge
0	1	0,1	1	Noninverted clock, sampled on falling edge
1	0	1,0	2	Inverted clock, sampled on rising edge
1	1	1,1	3	Inverted clock, sampled on falling edge
		Clock Sense		Description
		Noninverted		Signal is idle low, active high
		Inverted		Signal is idle high, active low

Peripheral manufacturers did not define a standard signaling convention in the beginning. Consequently SPI controllers often allow configuration of any of the four modes while the remaining only permit two of the modes. However, once a mode has been chosen, all slaves on the same bus must agree.

Signaling

The clock polarity determines the idle clock level, while the phase determines whether the data line is sampled on the rising or falling clock signal. Figure 16-2 shows mode 0,0, which is perhaps the preferred form of SPI signaling. In Figure 16-2, the slave is selected first, by making the $\overline{SS}$ (slave select) active. Only one slave can be selected at a time, since there must be one slave driving the MISO line. Shortly after the slave is selected, the master drives the MOSI line, and the slave simultaneously drives the MISO line with the first data bit. This can be the most or least significant bit, depending on how the controller is configured. The diagram shows the least significant bit first.

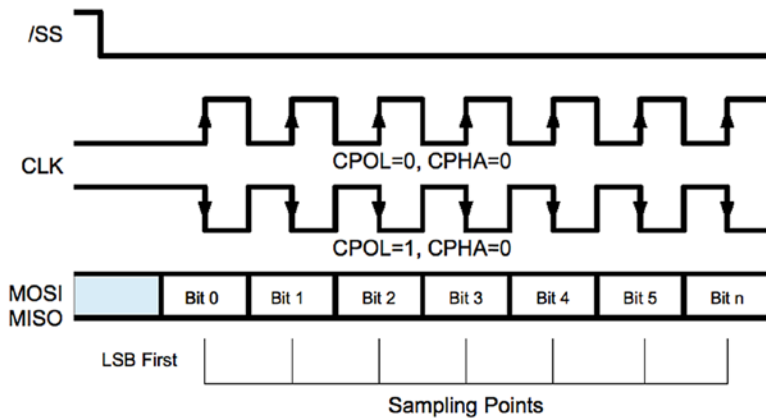

Figure 16-2. *SPI signaling, modes 0 and 2*

In mode 0,0 the first bit is clocked into the master and slave when the clock line falls from high to low. This clock transition is positioned midway in the data bit cell. The remaining bits are successively clocked into master and slave simultaneously as the clock transitions from high to low. The transmission ends when the master deactivates the slave select line. When the clock polarity is reversed (CPOL = 1, CPHA = 0), the clock signal shown in Figure 16-2 is simply inverted. The data is clocked at the same time in the data cell, but on the rising edge of the clock instead.

Figure 16-3 shows the clock signals with the phase set to 1 (CPHA = 1). When the clock is not inverted (CPOL = 0), the data is clocked on the rising edge. The clock must transition to its nonidle state one-half clock cycle earlier than when the phase is 0 (CPHA = 0). When the SPI mode is 1,1, the data is clocked in on the falling edge of the clock.

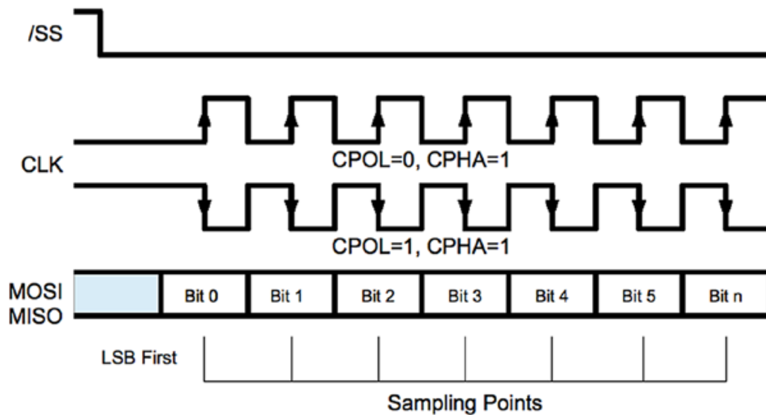

Figure 16-3. *SPI signaling modes 1 and 3*

While the four different modes can be confusing, it is important to realize that the data is sampled at the same times within the data bit cells. The data bit is always sampled at the midpoint of the data cell. When the clock phase is 0 (CPHA = 0), the data is sampled on the trailing edge of the clock, whether falling or rising according to CPOL. When the clock phase is 1 (CPHA = 1), the data is sampled on the leading edge of the clock, whether rising or falling according to CPOL.

Slave Selection

Unlike I²C where slaves are addressed by using a transmitted address, the SPI bus uses a dedicated select line for each. The Raspberry Pi dedicates the GPIO pins listed in Table 16-2 as slave select lines (also known as chip enable lines).

Table 16-2. *Raspberry Pi Built-in Chip Enable Pins*

GPIO	Chip Enable	P1
8	$\overline{CE0}$	P1-24
7	$\overline{CE1}$	P1-26

The Raspbian Linux kernel driver supports the use of only these two chip enable lines. However, the driver is designed such that you don't have to use them, or only these. It is possible, for example, to use a different GPIO pin as a select under user software control. The application simply takes responsibility for activating the slave select GPIO line prior to the data I/O and deactivates it after. When the driver is controlling the slave selects, this is done automatically.

Driver Support

To enable the SPI driver, edit the `/boot/config.txt` file to uncomment the line as:

```
dtparam=spi=on
```

and then reboot:

```
# sync
# /sbin/shutdown -r now
```

After the reboot, using the `lsmod` command, you should see the driver `spi_bcm2835` listed among the others.

```
$ lsmod
Module                  Size  Used by
fuse                  106496  3
rfcomm                 49152  6
...
spi_bcm2835            16384  0
...
```

Once the kernel module support is present, the device driver nodes should appear:

```
$ ls /dev/spi*
/dev/spidev0.0  /dev/spidev0.1
$
```

These two device nodes are named according to which slave select should be activated, as shown in Table 16-3.

Table 16-3. *SPI Device Nodes*

Pathname	Bus	Device	GPIO	$\overline{SS}$
/dev/spidev0.0	0	0	8	$\overline{CE0}$
/dev/spidev0.1	0	1	7	$\overline{CE1}$

If you open either of these device nodes with the C macro SPI_NO_CS, the node chosen makes no difference. Macro SPI_NO_CS indicates that slave select will be performed by the application instead of the driver, if any select is used at all. When only one slave device is attached, might it be possible to use a permanently hard-wired selected.

SPI API

Like I^{2}C under Linux, the bare-metal API for SPI involves calls to ioctl(2) to configure the interface and for simultaneous read/write. The usual read(2) and write(2) system calls can be used for one-sided transfers.

Header Files

The header files needed for SPI programming are as follows:

```
#include <fcntl.h>
#include <unistd.h>
#include <stdint.h>
#include <sys/ioctl.h>
#include <linux/types.h>
#include <linux/spi/spidev.h>
```

The `spidev.h include` file defines several macros and the `struct spi_ioc_transfer`. Table 16-4 lists the main macros that are declared. The macros `SPI_CPOL` and `SPI_CPHA` are used in the definitions of the values `SPI_MODE_x`. If you prefer, it is possible to use `SPI_CPOL` and `SPI_CPHA` in place of the mode macros.

Table 16-4. *SPI Macro Definitions*

Macro	Supported	Description
SPI_CPOL	Yes	Clock polarity inverted (CPOL = 1)
SPI_CPHA	Yes	Clock phase is 1 (CPHA = 1)
SPI_MODE_0	Yes	SPI Mode 0,0 (CPOL = 0, CPHA = 0)
SPI_MODE_1	Yes	SPI Mode 0,1 (CPOL = 0, CPHA = 1)
SPI_MODE_2	Yes	SPI Mode 1,0 (CPOL = 1, CPHA = 0)
SPI_MODE_3	Yes	SPI Mode 1,1 (CPOL = 1, CPHA = 1)
SPI_CS_HIGH	Yes	Chip select is active high
SPI_LSB_FIRST	No	LSB is transmitted first
SPI_3WIRE	No	Use 3-Wire data I/O mode
SPI_LOOP	No	Loop the MOSI/MISO data line
SPI_NO_CS	Yes	Do not apply Chip Select
SPI_READY	No	Enable extra Ready signal

Communicating with an SPI device consists of the following system calls:

`open(2)`: Opens the SPI device driver node

`read(2)`: Reads but no transmission

`write(2)`: Writes data while discarding received data

`ioctl(2)`: For configuration and bidirectional I/O

`close(2)`: Closes the SPI device driver node

In SPI communication, the use of `read(2)` and `write(2)` is generally unusual. Normally, `ioctl(2)` is used to facilitate simultaneous read/write transfers.

Open Device

In order to perform SPI communication through the kernel driver, you need to open one of the device nodes by using `open(2)`. The general format of the device pathname is

```
/dev/spidev<bus>.<device>
```

The following is a code snippet opening bus 0, device 0.

```
int fd;

fd = open("/dev/spidev0.0",O_RDWR);
if ( fd < 0 ) {
    perror("Unable to open SPI driver");
    exit(1);
}
```

The driver is normally opened for read and write (`O_RDWR`) because SPI usually involves reading and writing.

SPI Mode Macros

Before SPI communications can be performed, the mode of communication needs to be chosen. Table 16-5 lists the C language macros that can be used to configure the SPI mode to apply.

Table 16-5. *SPI Mode Macros*

Macro	Effect	Comments
SPI_CPOL	CPOL = 1	Or use SPI_MODE_x
SPI_CPHA	CPHA = 1	Or use SPI_MODE_x
SPI_CS_HIGH	SS is active high	Unusual
SPI_NO_CS	Don't assert select	Not used/application controlled

These bit values are simply or-ed together to specify the options that are required. The use of `SPI_CPOL` implies CPOL = 1. Its absence implies CPOL = 0. Similarly, the use of `SPI_CPHA` implies CPHA = 1 else CPHA = 0. The macros `SPI_MODE_x` use the `SPI_CPOL` and `SPI_CPHA` macros to define them, so don't use them both in your code. The mode definitions are shown here:

```
#define SPI_MODE_0 (0|0)
#define SPI_MODE_1 (0|SPI_CPHA)
#define SPI_MODE_2 (SPI_CPOL|0)
#define SPI_MODE_3 (SPI_CPOL|SPI_CPHA)
```

The unsupported options are not shown, though one or more of these could be supported in the future.

The following is an example that defines `SPI_MODE_0`:

```
uint8_t mode = SPI_MODE_0;
int rc;

rc = ioctl(fd,SPI_IOC_WR_MODE,&mode);
if ( rc < 0 ) {
    perror("Can't set SPI write mode.");
```

If you'd like to find out how the SPI driver is currently configured, you can read the SPI mode with `ioctl(2)` as follows:

```
uint8_t mode;
int rc;

rc = ioctl(fd,SPI_IOC_RD_MODE,&mode);
if ( rc < 0 ) {
     perror("Can't get SPI read mode.");
```

Bits per Word

The SPI driver needs to know how many bits per I/O word are to be transmitted. While the driver will likely default to 8 bits, it is best not to depend on it. Note that the Pi only supports 8 bits or 9 bits in LoSSI mode (low speed serial interface). This is configured with the following `ioctl(2)` call:

```
uint8_t bits = 8;
int rc;

rc = ioctl(fd, SPI_IOC_WR_BITS_PER_WORD,&bits);
if ( rc < 0 ) {
    perror ("Can't set bits per SPI word.");
```

The currently configured value can be fetched with `ioctl(2)` as follows:

```
uint8_t bits;
int rc;

rc = ioctl(fd,SPI_IOC_RD_BITS_PER_WORD,&bits);
if ( rc == -1 ) {
    perror("Can't get bits per SPI word.");
```

When the number of bits is not an even multiple of eight, the bits are assumed to be right-justified. For example, if the word length is set to 4 bits, the least significant 4 bits are transmitted. The higher-order bits are ignored.

Likewise, when receiving data, the least significant bits contain the data. All of this is academic on the Pi, however, since the driver supports only byte-wide transfers.

Clock Rate

To configure the data transmission rate, you can set the clock rate with `ioctl(2)` as follows:

```
uint32_t speed = 500000; /* Hz */
int rc;

rc = ioctl(fd,SPI_IOC_WR_MAX_SPEED_HZ,&speed);
if ( rc < 0 ) {
    perror("Can't configure SPI clock rate.");
```

The clock rate provided in speed should be a multiple of two (it is automatically rounded down). The current configured clock rate can be fetched using the following `ioctl(2)` call:

```
uint32_t speed; /* Hz */
int rc;

rc = ioctl(fd,SPI_IOC_RD_MAX_SPEED_HZ,&speed);
if ( rc < 0 ) {
    perror("Can't get SPI clock rate.");
```

Data I/O

SPI communication often involves transmitting data while simultaneously receiving data. For this reason, the `read(2)` and `write(2)` system calls cannot be used. The `ioctl(2)` call will, however, perform a simultaneous read and write.

The SPI_IOC_MESSAGE(n) form of the ioctl(2) call uses the following structure as its argument:

```
struct spi_ioc_transfer {
    __u64   tx_buf;        /* Ptr to tx buffer */
    __u64   rx_buf;        /* Ptr to rx buffer */
    __u32   len;           /* # of bytes */
    __u32   speed_hz;      /* Clock rate in Hz */
    __u16   delay_usecs;   /* Delay in microseconds */
    __u8    bits_per_word; /* Bits per "word" */
    __u8    cs_change;     /* Apply chip select */
    __u32   pad;           /* Reserved */
};
```

The tx_buf and rx_buf structure members are defined as 64-bit unsigned integers (__u64). For this reason, you must cast your buffer pointers when making assignments to them:

```
uint8_t tx[32], rx[32];
struct spi_ioc_transfer tr;

tr.tx_buf = (unsigned long) tx;
tr.rx_buf = (unsigned long ) rx;
```

On the Raspberry Pi, you will see example code that simply casts the pointers to unsigned long. The compiler automatically promotes these 32-bit values to a 64-bit value. This is safe on the Pi because the pointer value is 32 bits in size.

If you don't wish to receive data (maybe because it is "don't care" data), you can null out the receive buffer:

```
uint8_t tx[32];
struct spi_ioc_transfer tr;

tr.tx_buf = (unsigned long) tx;
tr.rx_buf = 0;                  /* ignore received data */
```

Note that to receive data, the master must always transmit data in order to shift data out of the slave peripheral. If any byte transmitted will do, you can omit the transmit buffer. Zero bytes will then be automatically transmitted by the driver to shift the slave data out.

It is also permissible to transmit from the buffer you're receiving into:

```
uint8_t io[32];
struct spi_ioc_transfer tr;

tr.tx_buf = (unsigned long) io;         /* Transmit buffer */
tr.rx_buf = (unsigned long) io;         /* is also recv buffer */
```

The `len` structure member indicates the number of bytes for the I/O transfer. Receive and transmit buffers (when both used) are expected to transfer the same number of bytes.

The member `speed_hz` defines the clock rate that you wish to use for this I/O, in Hz. This overrides any value configured in the mode setup, for the duration of the I/O. The value will be automatically rounded down to a supported clock rate when necessary.

When the value `speed_hz` is 0, the previously configured clock rate is used (`SPI_IOC_WR_MAX_SPEED_HZ`).

When the `delay_usecs` member is non-zero, it specifies the number of microseconds to delay between transfers. It is applied at the end of a transfer, rather than at the start. When there are multiple I/O transfers in a single `ioctl(2)` request, this allows time in between so that the peripheral can process the data.

The `bits_per_word` member defines how many bits there are in a "word" unit. Often the unit is 1 byte (8 bits), but it need not be (but note that the Raspbian Linux driver supports only 8 bits or 9 in LoSSI mode).

When the `bits_per_word` value is 0, the previously configured value from `SPI_IOC_WR_BITS_PER_WORD` is used.

The `cs_change` member is treated as a Boolean value. When 0, no chip select is performed by the driver. The application is expected to do what is necessary to notify the peripheral that it is selected (usually a GPIO pin is brought low). Once the I/O has completed, the application then must then unselect the slave peripheral.

When the `cs_change` member is `true` (non-zero), the slave selected will *depend on the device pathname that was opened*. The bus and the slave address are embedded in the device name:

```
/dev/spidev<bus>.<device>
```

When `cs_change` is `true`, the driver asserts $\overline{GPIO8}$ for `spidev0.0` and asserts $\overline{GPIO7}$ for `spidev0.1` prior to I/O and then deactivates the same upon completion. Of course, using these two nodes require two different `open(2)` calls.

The `SPI_IOC_MESSAGE(n)` macro is used in the `ioctl(2)` call to perform one or more SPI I/O operations. This macro is unusual because it requires an argument *n*. (This differs considerably from the I2C approach.) This specifies how many I/O transfers you would like to perform. An array of `spi_ioc_transfer` structures is declared and configured for each transfer required, as shown in the next example:

```
struct spi_ioc_transfer io[3];    /* Define 3 transfers */
int rc;

io[0].tx_buf = ...;               /* Configure I/O */
...
io[2].bits_per_word = 8;

rc = ioctl(fd,SPI_IOC_MESSAGE(3),& io[0]);
```

The preceding example will perform three I/O transfers. Since the application never gets to perform any GPIO manipulation in between these I/Os, this applies to communicating with one particular slave device.

The following example code brings all of the concepts together, to demonstrate one I/O. The `spi_ioc_transfer` structure is initialized so that 32 bytes are transmitted and simultaneously 32 are received.

```
uint8_t tx[32], rx[32];
struct spi_ioc_transfer tr;
int rc;

tr.tx_buf        = (unsigned long) tx;
tr.rx_buf        = (unsigned long) rx;
tr.len           = 32;
tr.delay_usecs   = delay;
tr.speed_hz      = speed;
tr.bits_per_word = bits;

rc = ioctl(fd,SPI_IOC_MESSAGE(1),&tr);
if ( rc < 1 ) {
    perror("Can't send spi message");
```

Here a single I/O transmission occurs, with data being sent from array `tx` and received into array `rx`. The return value from the `ioctl(2)` call returns the number of bytes transferred (32 in the example). Otherwise, `-1` is returned to indicate that an error has occurred.

Close

Like all Unix I/O operations, the device must be closed when the open file descriptor is no longer required (otherwise it will be done upon process termination):

```
close(fd);
```

Write

The `write(2)` system call can be used if the received data is unimportant. Note, however, that no delay is possible with this call.

Read

The `read(2)` system call is actually inappropriate for SPI since the master must transmit data on MOSI in order for the slave to send bits back on the MISO line. However, when `read(2)` is used, the driver will automatically send out zero bits as necessary to accomplish the read. (Be careful that your peripheral will accept zero bytes without unintended consequences.) Like the `write(2)` call, no delay is possible.

SPI Testing

When developing your SPI communication software, you can perform a simple loopback test to test your framework. Once the framework checks out, you can then turn your attention to communicating with the actual device.

While the `SPI_LOOP` mode bit is not supported by the Pi hardware, you can still physically loop your SPI bus by connecting a wire from the MOSI output back to the MISO input pin (connect GPIO 10 to GPIO 9).

A simple program, shown next, demonstrates this type of loopback test. It will write out 4 bytes (0x12, 0x23, 0x45, and 0x67) to the SPI driver. Because you have wired the MOSI pins to the MISO input, anything transmitted will also be received.

When the program executes, it will report the number of bytes received and four hexadecimal values:

```
$ sudo ./spiloop
rc=4 12 23 45 67
$
```

If you remove the wire between MOSI and MISO, and connect the MISO to a high (+3.3 V), you should be able to read 0xFF for all of the received bytes. If you then connect MISO to ground, 0x00 will be received for each byte instead. Be certain to apply to the correct pin to avoid damage (Listing 16-1).

Listing 16-1. The spiloop.c SPI loopback program

```
/*********************************************
 * spiloop.c – Example loop test
 * Connect MOSI (GPIO 10) to MISO (GPIO 9)
 *********************************************/
0005: #include <stdio.h>
0006: #include <errno.h>
0007: #include <stdlib.h>
0008: #include <stdint.h>
0009: #include <fcntl.h>
0010: #include <unistd.h>
0011: #include <sys/ioctl.h>
0012: #include <linux/types.h>
0013: #include <linux/spi/spidev.h>
0014:
0015: static void
0016: errxit(const char *msg) {
0017:   perror(msg);
0018:   exit(1);
0019: }
0020:
0021: int
0022: main(int argc, char ** argv) {
0023:   static uint8_t tx[] = {0x12, 0x23, 0x45, 0x67};
0024:   static uint8_t rx[] = {0xFF, 0xFF, 0xFF, 0xFF};
0025:   struct spi_ioc_transfer ioc = {
```

```
0026:         .tx_buf = (unsigned long) tx,
0027:         .rx_buf = (unsigned long) rx,
0028:         .len = 4,
0029:         .speed_hz = 100000,
0030:         .delay_usecs = 10,
0031:         .bits_per_word = 8,
0032:         .cs_change = 1
0033:     };
0034:     uint8_t mode = SPI_MODE_0;
0035:     int rc, fd=-1;
0036:
0037:     fd = open("/dev/spidev0.0",O_RDWR);
0038:     if ( fd < 0 )
0039:         errxit("Opening SPI device.");
0040:
0041:     rc = ioctl(fd,SPI_IOC_WR_MODE,&mode);
0042:     if ( rc < 0 )
0043:         errxit("ioctl (2) setting SPI mode.");
0044:
0045:    rc = ioctl(fd,SPI_IOC_WR_BITS_PER_WORD,&ioc.bits_per_word);
0046:     if ( rc < 0 )
0047:         errxit("ioctl (2) setting SPI bits perword.");
0048:
0049:     rc = ioctl(fd,SPI_IOC_MESSAGE(1),&ioc);
0050:     if ( rc < 0 )
0051:         errxit("ioctl (2) for SPI I/O");
0052:     close(fd);
0053:
0054:     printf("rc=%d %02X %02X %02X %02X\n",
0055:         rc, rx[0], rx[1], rx[2], rx[3]);
0056:     return 0;
0057: }
```

Summary

The SPI bus and its operation were presented along with the C programming API. The chapter ended with a simple SPI loop test program. No extra hardware was required to run this.

That loop test provides a good coverage of the API being applied. The reader can take this one step further and access an actual slave device on the SPI bus. That last step adds the slave select to the overall picture and any command/response processing required of the device. You are now the SPI master!