SPI Guide for Linux C on BeagleBone Green

Authors:
Ian Cruikshank
Ryan Garofano
Rajnesh Joshi
Jaydon Vanloo

December 7, 2022

We have yet to find any guide or learning material about using SPI in Linux that is comprehensive and easy to follow specifically for the BeagleBone Green. Our aim is for this guide to be just that.

CONTENTS

1 BUS DESCRIPTION

The SPI (Serial Peripheral Interface) protocol is for synchronously communicating between a controller/master and any number of peripheral/slave devices. The SPI interface supports full-duplex transferring, meaning data is sent and received on different pins. In contrast, the I2C interface only has 1 data pin, so it can only support half-duplex transferring.

Every SPI interface will have the following pins:

Name	Description
CS	Chip Select. The Peripheral/Slave has 1 CS pin, while the Controller/Master has any number of CS pins, CS0, CS1, etc.
SCLK	Clock
MOSI/COPI	Master-Out-Slave-In, or Controller-Out-Peripheral-In
MISO/CIPO	Master-In-Slave-Out, or Controller-In-Peripheral-Out

Every pin in the interface is driven by the controller except MISO, which is the data sent to the controller by the peripheral.

The number of CS pins an SPI bus has equals the number of peripherals that can be connected to that bus. Every peripheral on an SPI bus connects to the same data and clock pins, but each peripheral connects to a different chip select pin. Below is a simple diagram to demonstrate.

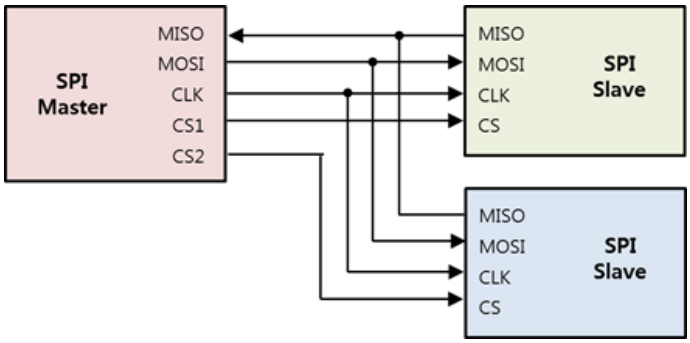

Figure 1: SPI Multi-Peripheral Diagram.
From https://docs.tizen.org/iot/guides/peripheral-io-api-spi/

A peripheral is only enabled to communicate on its bus when its CS pin is low. When it is high, the peripheral ignores its interface's data and clock pins. This is how it is possible to connect multiple peripherals to the same data pins and still be able to communicate with each of them separately. When the controller is to send/receive from a particular peripheral, it sets that peripheral's CS pin low. Otherwise, all CS pins will be set high.

2 BeagleBone SPI Buses

The BeagleBone Green has 2 SPI buses, SPI0 and SPI1. SPI0 has 1 CS pin and SPI1 has 2 CS pins. Since SPI1 has 2 CS pins, the available connections will appear as 2 different devices.

HW bus	Linux Device	Pins
SPI0	/dev/spidev0.0	CS: P9_17 (SPI0_CS0) SCLK: P9_22 (SPI0_SCLK) MOSI: P9_18 (SPI0_D1) MISO: P9_21 (SPI0_D0)
SPI1, CS0	/dev/spidev1.0	CS: P9_28 (SPI1_CS0), or P9_20 SCLK: P9_31 (SPI1_SCLK) MOSI: P9_30 (SPI1_D1) MISO: P9_29 (SPI1_D0)
SPI1, CS1	/dev/spidev1.1	CS: P9_42 (SPI1_CS1), or P9_19 Other pins are the same as for /dev/spidev1.0

3 Transaction Structure

SPI interfaces generally send and receive single bytes at a time. They support full-duplex transferring (sending and receiving simultaneously), but reading and writing has a 1-byte delay. The diagram below shows the beginning of a general SPI transaction.

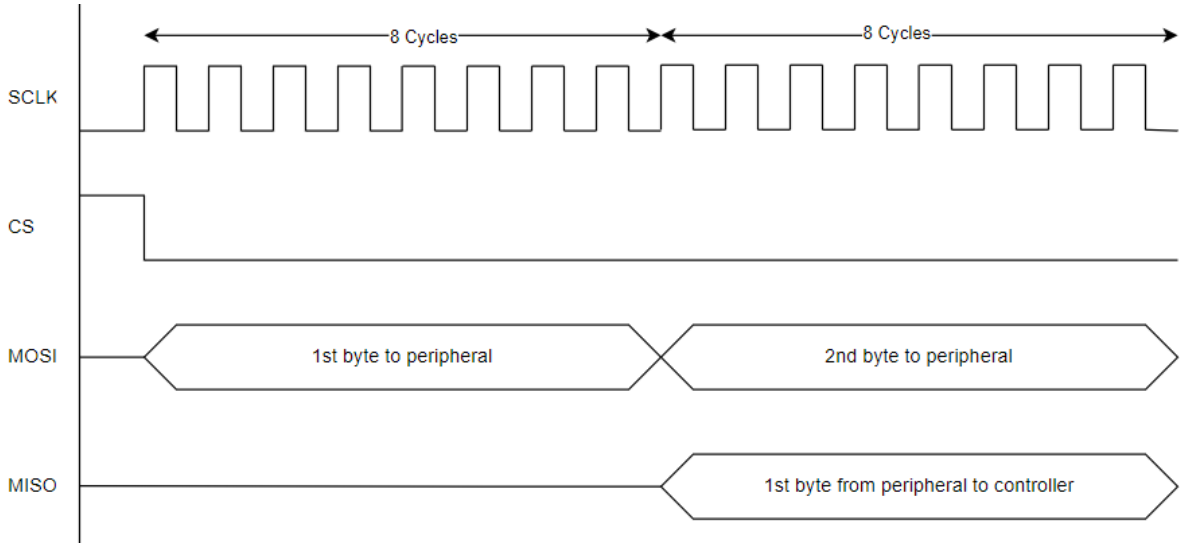

Figure 2: Beginning of a General SPI Transaction

Notice that the controller is sending its second byte to the peripheral at the same time that the peripheral is sending its first byte back to the controller. The first byte sent back to the controller comes 8 cycles or 1 byte after the first byte sent to the peripheral. The MISO line is ignored during the first byte sent to the peripheral on the MOSI line. The total transaction is 2 bytes long after only the first byte is sent back to the controller. This means that if the controller wants to read N bytes from the peripheral, the total transaction must be N + 1 bytes long.

For example, imagining a device that supports consecutive reads from different registers, a read transaction of 2 registers in a row might look like the following.

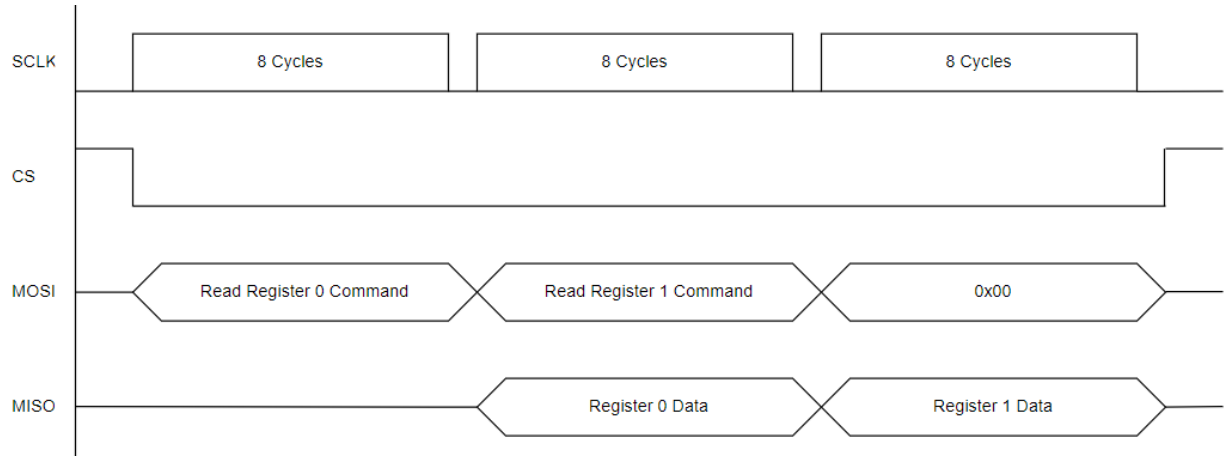

Figure 3: Consecutive Register Read Example

The device register data comes on the MISO line 1 byte after the read command is sent on the MOSI line. So, the total transaction length is 1 byte longer than the number of bytes read from the peripheral and the MISO line is ignored during the first byte sent on the MOSI line.

4 SPI VIA C CODE

4.1 INITIALIZATION

IMPORTANT PREFACE: Any program that uses SPI on the BeagleBone MUST be started by the root or super user. If you encounter SPI bus access errors, run the program with **sudo**.

To initialize an SPI port on the BeagleBone, the following things need to be done:

1. **Configure the desired pins for SPI (Done in Command line or otherwise)**
   E.g. for SPI1, CS0 (command line):

   ```
   $ config-pin P9_28 spi_cs
   $ config-pin P9_31 spi_sclk
   $ config-pin P9_29 spi
   $ config-pin P9_30 spi
   ```

2. **Open the device file and get the file descriptor (in C program)**
   For this, the C file must include <fcntl.h>.
   E.g. for SPI1, CS0:

   ```
   int spiFileDesc = open("/dev/spidev1.0", O_RDWR);
   ```

3. **Set the SPI Mode**
   The mode of an SPI bus refers to when within each clock cycle data is latched and driven. This is

the only parameter that is absolutely necessary to set. The BeagleBone supports all 4 modes, so the choice is arbitrary but it does need to be set. A default of 0 is fine.
For this, the C file must include <ioctl.h> and <linux/spi/spidev.h>.

E.g.:

```
int spiMode = 0;
ioctl(spiFileDesc, SPI_IOC_WR_MODE, &spiMode);
```

Optionally, other bus parameters can be set but the mode is the only one required.

Below is a sample bus initialization function written in C, combining the last 2 steps above. It assumes pins are already configured. It also does an optional setting of the max speed of the bus.

```
#include <fcntl.h>
#include <linux/spi/spidev.h>

// Assume pins already configured for SPI
// E.g. for SPI1, CS0:
// (bbg)$ config-pin P9_28 spi_cs
// (bbg)$ config-pin P9_31 spi_sclk
// (bbg)$ config-pin P9_29 spi
// (bbg)$ config-pin P9_30 spi

#define SPI_DEV_BUS0_CS0 "/dev/spidev0.0"
#define SPI_DEV_BUS1_CS0 "/dev/spidev1.0"
#define SPI_DEV_BUS1_CS1 "/dev/spidev1.1"

#define SPI_MODE_DEFAULT 0
#define SPEED_HZ_DEFAULT 500000 // Arbitrary, but 500000 is reasonable

int SPI_initPort(char* spiDevice)
{
    // Open Device
    int spiFileDesc = open(spiDevice, O_RDWR);
    if (spiFileDesc < 0) {
        printf("Error: Can't open device %s\n", spiDevice);
        exit(1);
    }

    // Set port parameters

    // Set SPI mode: Necessary
    int spiMode = SPI_MODE_DEFAULT;
    int errorCheck = ioctl(spiFileDesc, SPI_IOC_WR_MODE, &spiMode);
    if (errorCheck < 0) {
        printf("Error: Set SPI mode failed\n");
        exit(1);
    }

    // Set Max Speed (Hz): Optional
    int speedHz = SPEED_HZ_DEFAULT;
    errorCheck = ioctl(spiFileDesc, SPI_IOC_WR_MAX_SPEED_HZ, &speedHz);
    if (errorCheck < 0) {
        printf("Error: Set max speed failed\n");
        exit(1);
    }

    return spiFileDesc;
}
```

4.2 SPI TRANSFERS

We want to be able to send an array of bytes on the MOSI line and receive an array of bytes on the MISO line. I.e. we want to be able to send commands and receive data from a generic device.

Recall the transaction in *Figure 3*, where register read commands were sent to an imaginary SPI device and the register data was returned with a 1-byte delay. Here we will demonstrate how to perform that transfer in C. We assume that the SPI bus is already initialized and that the file descriptor is available.

1. Allocate the buffers to send and receive. The transaction in *Figure 3* consisted of 2 register read commands, and the data returned is delayed by 1 byte, so the total transaction is 3 bytes long. Therefore, allocate 3 bytes for the buffer to send *and* for the buffer to receive.

   ```
   uint8_t *sendBuf = malloc(3);
   uint8_t *receiveBuf = malloc(3);
   ```

2. Set the commands in the buffer to send. Only the first two bytes of the buffer to send are read commands. The third byte should be the cancel byte: it should tell the device you're reading from that the command before it was the last read. This will depend on the device. We assume for our imaginary device that it should be 0. The command to read registers will also depend on the device. This is up to you to find. It will be in the device's data sheet and will probably contain the register address and a bit indicating whether the operation is a read or a write. Again, our device is imaginary.

   ```
   sendBuf[0] = READ_REGISTER_0_COMMAND;
   sendBuf[1] = READ_REGISTER_1_COMMAND;
   sendBuf[2] = 0x00;
   ```

3. Create an instance of struct **spi_ioc_transfer** provided in <linux/spi/spidev.h>. Use **memset** in <string.h> to zero out the entire struct before setting any of its fields. We don't need to set all of its fields and the ones we don't need to set may end up being garbage when the struct is created, which may affect the transaction.

   ```
   struct spi_ioc_transfer transfer;
   memset(&transfer, 0, sizeof(transfer));
   ```

4. Set the 3 necessary fields in the struct, **tx_buf**, **rx_buf**, and **len**, which are a pointer to the buffer to send, a pointer to the buffer to receive, and the length of both in bytes, respectively. **tx_buf** and **rx_buf** are pointers to the buffers, but their types in the struct are actually unsigned longs, so do a cast.

   ```
   transfer.tx_buf = (unsigned long) sendBuf;
   transfer.rx_buf = (unsigned long) receiveBuf;
   transfer.len = 3;
   ```

5. Make an ioctl call to perform the actual transfer. This ioctl call will use the SPI device file descriptor, a pointer to the transfer struct, and macro from <linux/spi/spidev.h> which takes the number of structs being transferred. We are only using 1 transfer struct so this will just be 1. Optionally, the pointer to the transfer struct could instead be a pointer to an array of transfer

structs, but generally only 1 is needed because the send and receive buffers can be of any size.

```
const int NUM_TRANSFERS = 1;
int status = ioctl(spiFileDesc, SPI_IOC_MESSAGE(NUM_TRANSFERS),
&transfer);
if (status < 0) {
    printf("Error: SPI Transfer failed\n");
}
```

6. Now that the transfer has been performed, we can extract the register data. Recall in figure 3 how the returned data is delayed by 1 byte. Therefore, we ignore the first byte in receiveBuf. The second byte will be the register 0 data and the third byte will be the register 1 data.

   ```
   uint8_t reg0Data = receiveBuf[1];
   uint8_t reg1Data = receiveBuf[2];
   ```

   The transaction is done.

7. Troubleshooting:
   If you encounter any unexpected results or errors, make sure that you are using **memset** in <string.h> to zero out the transfer struct before assigning anything, OR that you are assigning every field. The fields in the struct could potentially be garbage since we don't need to set all of them, and they may affect the transaction.

   Make sure the buffer to send and the buffer to receive are the *same* length.

   If no errors are occurring but reading from registers for example is returning seemingly random values, set the bus speed to 500 000 Hz. It should be at a working value by default, but for some devices, the max speed BeagleBone supports is too high. 500 000 Hz is a reasonable speed that shouldn't be too fast for any device.

Here is a sample SPI transfer function which performs steps 3, 4, and 5 from above.

```
void SPI_transfer(int spiFileDesc, uint8_t *sendBuf, uint8_t *receiveBuf, int length)
{
    // Setting transfer this way ensures all other fields are 0
    struct spi_ioc_transfer transfer = {
        .tx_buf = (unsigned long) sendBuf,
        .rx_buf = (unsigned long) receiveBuf,
        .len = length
    };

    const int NUM_TRANSFERS = 1;
    int status = ioctl(spiFileDesc, SPI_IOC_MESSAGE(NUM_TRANSFERS), &transfer);
    if (status < 0) {
        printf("Error: SPI Transfer failed\n");
    }
}
```