

How to Access Chips Over the SPI

The [BeagleBone Black](#) is a Single Board Computer for \$45 which runs an ARM CPU, 2Gb of flash, 512Mb of RAM and comes with many connectors allowing you to interface with electronics. Unlike the Arduino, the BeagleBone Black runs a full Linux kernel, allowing you to talk directly to your electronics from the language of your choice and with the comfort of all that RAM. In my previous article I looked at how to talk to the GPIO pins on the BeagleBone Black. This time around I'm stepping it up to talk to persistent storage in the form of an EEPROM over the Serial Peripheral Interface Bus ([SPI](#)) on the BeagleBone Black.

The SPI allows data to move in both directions from a bus master (controller) to various chips which are attached to the bus.

Because there are many chips on the bus, you need to have some way to stop chips talking at the same time or reacting to commands you have sent for another chip on the bus. So each chip gets its own Chip Select line which tells it that it is the active chip and you want to talk to it (and maybe hear from it).

The [normal interface](#) might have a wire to send data from the master to the bus, a wire to get data back from the bus, a clock wire to control when a new bit is sent and received, and a chip select wire (for each chip on the bus). The wire to send from the master to the chip (slave) is called the MOSI (Master Out, Slave In) and the converse for communication from the chip (Slave) is the MISO (Master In). You send every byte a single bit at a time. To send a bit you set the MOSI line to the current bit you want to send and use the clock line to tell the chip to grab the current bit. Then send the next bit the same way and so on. Each time you send a bit, you run one clock and the chip might also send you one bit back on the MISO line.

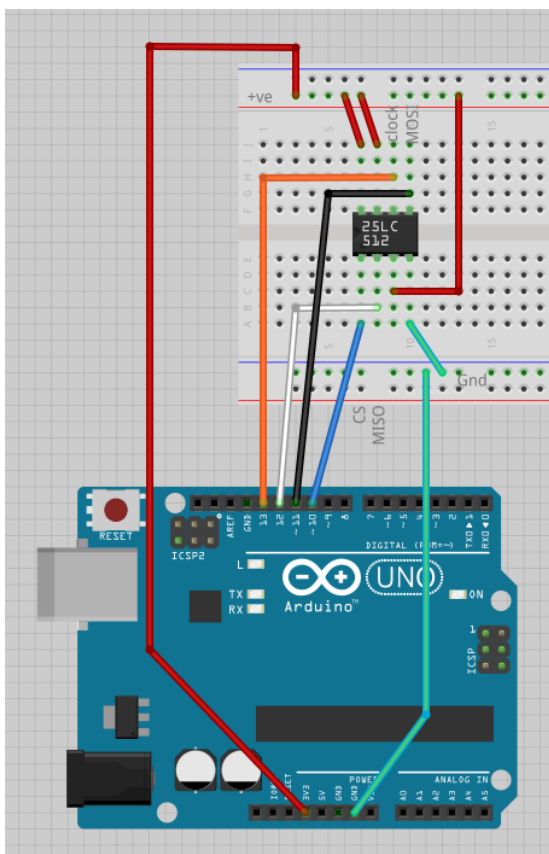
Testing the EEPROM with Arduino

A cheap EEPROM that has an SPI interface allows for testing SPI access without the risk of damaging more expensive hardware.

I choose a [512KBit EEPROM](#) which I got for a few dollars as the first SPI device. If you are worried about the speed of sending things around a bit at a time, that EEPROM can run at up to 20Mhz clock speed. The speed of sending the data a bit at a time might not be the slowest part of the system, for example EEPROM chips take time to write data to permanent storage.

I decided to access the EEPROM from Arduino first for two reasons: the Arduino board is cheaper than a BeagleBone Black, and I could also test that how I thought the EEPROM worked from reading the datasheet was how it worked when connected and powered on. Knowing that I can read and write to the EEPROM over SPI from an Arduino was useful when I need to debug issues accessing the EEPROM from the BeagleBone Black later.

While there are 8 pins on the EEPROM, half of those are connected to power rails (3 to the 3.3v supply and one to ground). I've shown all power wires in red, and the light blue wires are ground wires (below). The clock wire is shown in orange, the chip select in blue, and the MOSI and MISO in black and white respectively.



The Arduino program to read and write to the EEPROM is shown below. As you send data over the wire one bit at a time, you need to know what order the bits are to be sent in: that is, should you send the Most Significant Bit (MSB) first or last? The datasheet for the EEPROM stipulates that it wants the MSB first so the `setup()` function ensures that is the order. The chip select pin (10) is also set for output so we can talk to the EEPROM.

The main `loop()` of the program is quite simple, it just reads a byte from address 10, shows what it read to the user, and then writes the current loop iteration count to the byte at address 10. To read a byte from the EEPROM the READ instruction is sent, followed by the 2 byte address which you wish to read from. One might wonder about the bold `transfer(0)` line that follows the read and address, it seems that we should be reading the byte at that stage, not writing a zero. That `transfer(0)` call is used to send one byte, which will clock the SPI 8 times and thus have a byte of data for us from the chip. It doesn't matter what data we sent over the SPI after the read and address have been sent, the main thing is to clock the SPI so that the chip can send back the data we want. The reading will stop once we release the chip select line, and then we have to send another instruction to the EEPROM. To write to the EEPROM you send the write enable (INSTWREN) instruction, then a WRITE instruction, the address to write the data to, and then the byte to write. After that I write disable the IC again for completeness (INSTWRDI). You have to release the chip select after sending INSTWREN for that instruction to be acted on by the EEPROM. If you left the chip select LOW after writing INSTWREN and moved right on to issue the WRITE command, the EEPROM would likely ignore your write.

```
#include <SPI.h>
const byte INSTREAD = B0000011;
const byte INSTWRITE = B0000010;
const byte INSTWREN = B0000110;
```

```

const byte INSTWRDI = B0000100;
const int chipSelectPin = 10;
byte loopcount = 1;
void setup()
{
  Serial.begin(9600);
  // start the SPI library:
  SPI.begin();
  SPI.setBitOrder( MSBFIRST );
  pinMode(chipSelectPin, OUTPUT);
}
void loop()
{
  byte data = 0;
  digitalWrite(chipSelectPin, LOW);
  // Read the byte at address 10
  SPI.transfer(INSTREAD);
  SPI.transfer(0);
  SPI.transfer(10);
  data = SPI.transfer(0); // <- clock SPI 8 bits
  digitalWrite(chipSelectPin, HIGH);
  // show the user what we have
  Serial.print("read data:");
  Serial.print( data, BIN );
  Serial.print("\n");

  // Set write enable, write a byte with the current loop
  // and disable write again
  digitalWrite(chipSelectPin, LOW);
  SPI.transfer(INSTWREN);
  digitalWrite(chipSelectPin, HIGH);
  digitalWrite(chipSelectPin, LOW);
  SPI.transfer(INSTWRITE);
  SPI.transfer(0);
  SPI.transfer(10);
  SPI.transfer(loopcount);
  digitalWrite(chipSelectPin, HIGH);
  digitalWrite(chipSelectPin, LOW);
  SPI.transfer(INSTWRDI);
  digitalWrite(chipSelectPin, HIGH);

  loopcount++;
  delay(5000);
}

```

I should also mention that the EEPROM can give and get more data for each READ and WRITE instruction you send. For example, I could have clocked the SPI 8 times again to read the byte at address 11 without needing to issue another READ instruction. There are other capabilities to the EEPROM too, but for the purposes of the article the EEPROM is just used to check that one can read and write data over SPI.