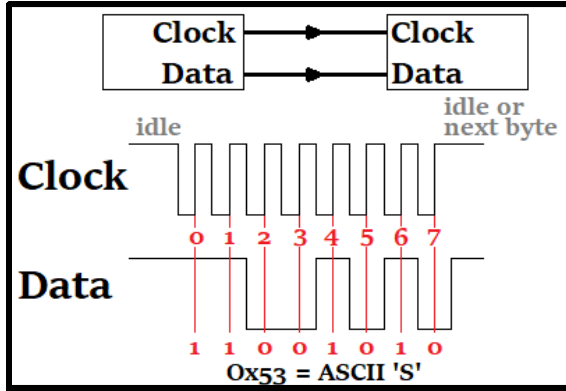
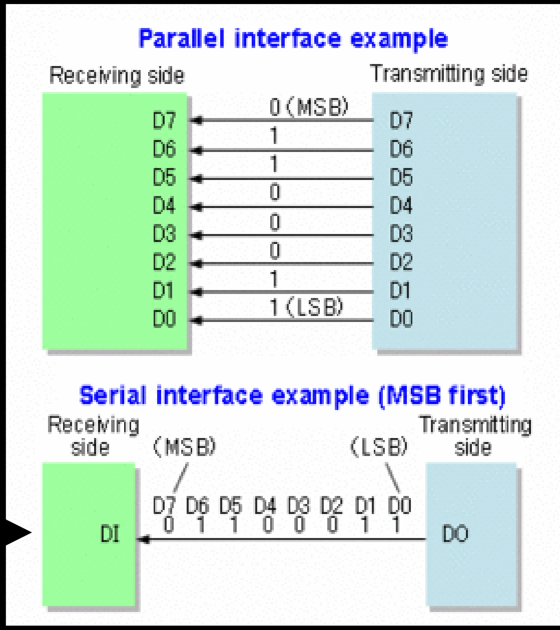
Introduction to SPI and Arduino

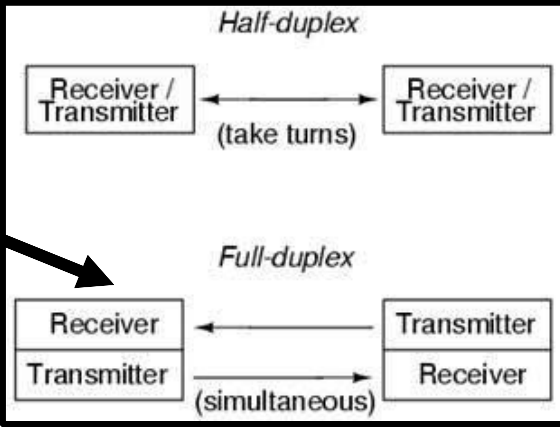
What is SPI?

A SPI (**S**erial **P**eripheral **I**nterface) is a method of communication, typically between a microcontroller and one or more peripheral devices (e.g. sensors).

- It is **synchronous**, meaning that the bits of data that are communicated are **synchronised with a clock signal**.
- It is **serial**, meaning that the **data is transferred one bit at a time**.
- And it is **full-duplex**, meaning that there are **two data lines**: one for sending data, and one for receiving data.







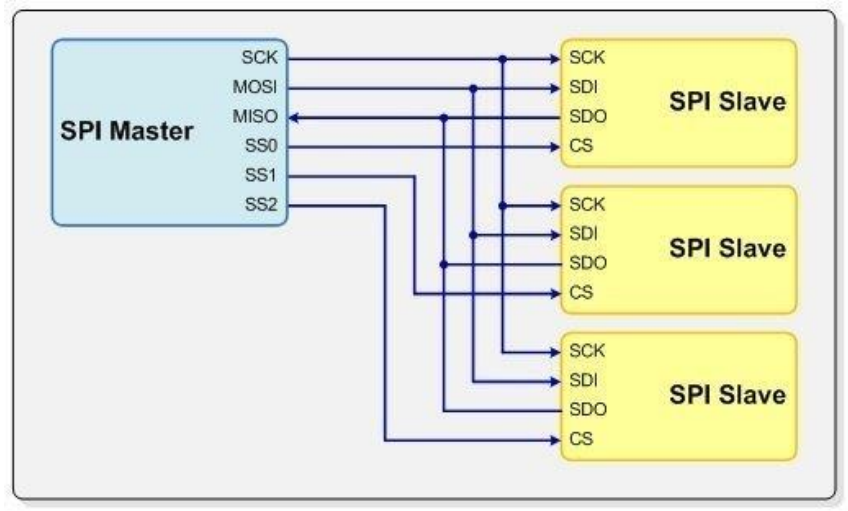
Useful terminology

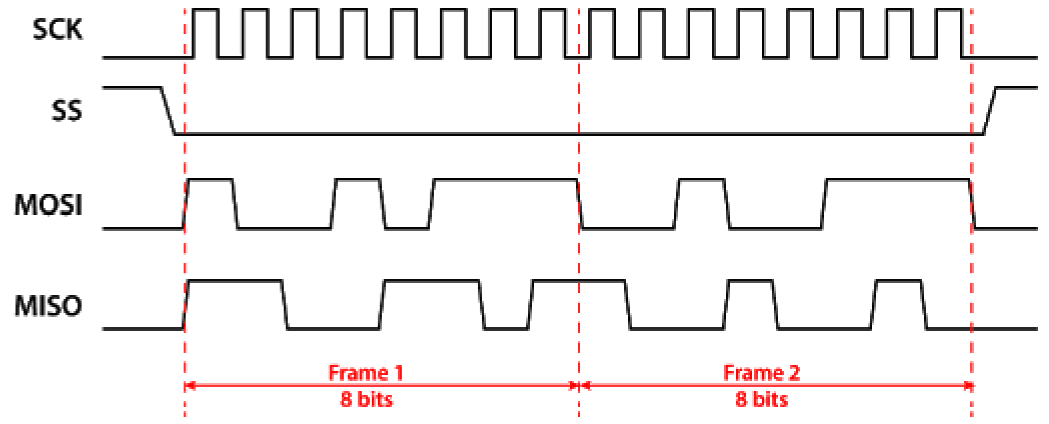
- SPI Bus | The ‘bus’ is just the ‘network’ of devices. If we had two sensors connected to the SPI of a microcontroller, we could say ‘there are three devices on the SPI bus’
- Master | SPI uses a ‘master’ and some ‘slaves’. The master device is the one that is in control of the bus. It is in charge of the clock signal, and it is the one that talks to the slaves and asks them for information (e.g. sensor outputs)
- Slave | and then the ‘slave’ devices just listen to the master, and send data to the master when it asks them to

How does it work?

SPI uses 4 wires in total.

- SCLK (Serial CLocK) – the 'synchronising' clock signal. This is used to synchronise the data transfer.
- MOSI (Master Out, Slave In) – the data transfer from master (the microcontroller), to slave (the peripheral device).
- MISO (Master In, Slave Out) – the data transfer from the slave, back to the master.
- SS (Slave Select) – a method of choosing which device on the SPI bus to communicate with. Each slave has a SS pin, and connects to a unique pin on the master. When the master wants to talk to a particular slave, it will take the corresponding SS pin for that device to logic LOW.





The setup process

If you want to set up an SPI bus for your application, there is some setup that needs to be done. There are quite a few parameters, and the ones you'll actually need vary between applications. In the SunSat case, we're not doing anything super complicated with the SPI bus, and so we only really need three settings.

Details for each of these settings will be given on following slides.

- **Clock Rate** – this is the rate at which pulses are sent on the SCLK line.
- **Bit Order** – this is about the order that the data is transferred in, and can be **Most Significant Bit First**, or **Least Significant Bit First**.
- **Data Mode** – SPI has 4 different modes of operation. They're a bit confusing, but basically just define at which point we send and receive data relative to clock cycles (e.g. rising or falling edge).

Clock rate

This one is nice and easy. **The clock rate is just the frequency at which the pulses on the SCLK line are sent.**

Each device will state a '**maximum SPI clock input frequency**' on it's datasheet. To the right, you can see a table from page 44 of the BMP388 datasheet that outlines some of the SPI timing settings.

The top line tells us that the BMP388 can handle a frequency on the SCLK line of up to 10MHz

Other devices may vary here, but for ease of use, it's usually best to keep the clock rate at a constant value that satisfies the constraints of all devices on the bus. For example, if one device has a maximum of 10MHz, and another has a maximum of 15Mhz, then set the clock rate as 10MHz.

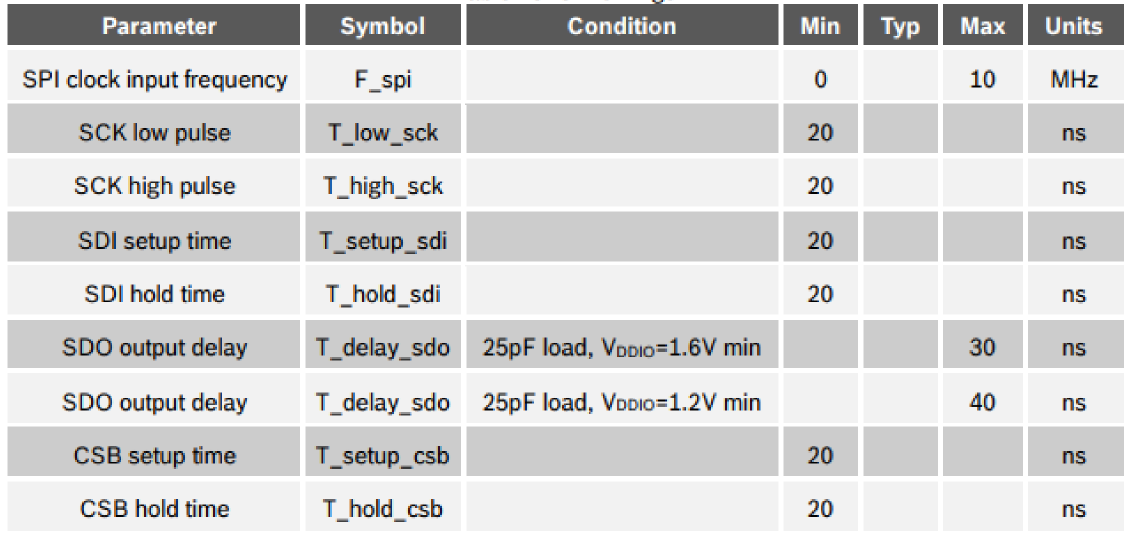


Table 48: SPI timings

Parameter	Symbol	Condition	Min	Typ	Max	Units
SPI clock input frequency	F_spi		0		10	MHz
SCK low pulse	T_low_sck		20			ns
SCK high pulse	T_high_sck		20			ns
SDI setup time	T_setup_sdi		20			ns
SDI hold time	T_hold_sdi		20			ns
SDO output delay	T_delay_sdo	25pF load, V_{DDIO}=1.6V min			30	ns
SDO output delay	T_delay_sdo	25pF load, V_{DDIO}=1.2V min			40	ns
CSB setup time	T_setup_csb		20			ns
CSB hold time	T_hold_csb		20			ns

Bit order

If I wanted to transfer the number *'11'* – which is *00001011* in binary – I could do so in the order of **Most Significant Bit First**, or **Least Significant Bit First**. The '1' bit at the end of the string is the least significant bit (LSB), and the '0' bit at the start of the string is the most significant bit (MSB).

Each device on the bus will be pre-configured into one of these orders. You can find the information on device datasheets. For example, the BMP388 datasheet has information about SPI configuration on page 41, and includes the figure to the right:

On the 'SDO' (serial data out) line at the bottom, you can see that bit *DO7* is the first one to be sent. In an 8-bit string, *DO7* is the most significant bit, and so we can see that the BMP388 is configured for **MSB First** communication.

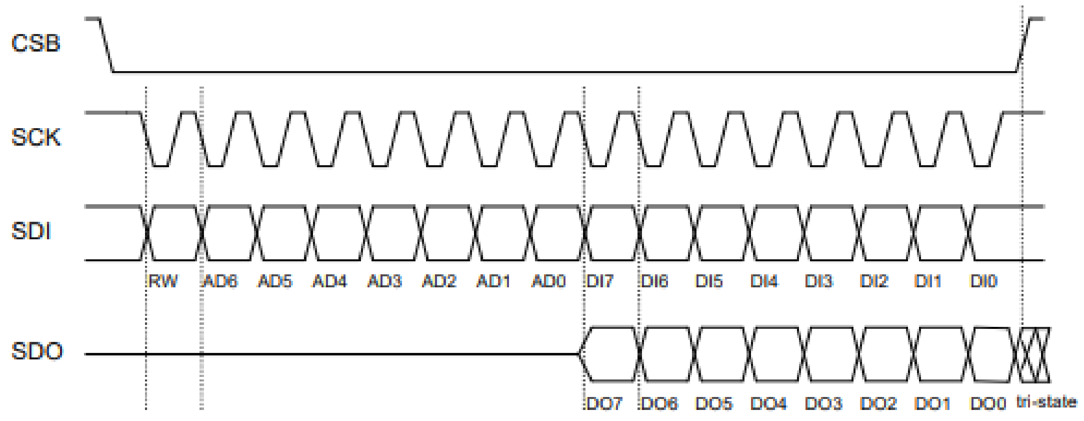


Figure 17: SPI protocol (shown for mode '11' in 4-wire configuration)

The communication process

Now, lets say we want to actually start communicating with a device on the bus. There is a relatively simple set of steps that we have to go through to do so. It differs very slightly between reading and writing, but I'll highlight the read process here.

Read data from slave

1. For the slave we want to read from, pull the relevant SS pin to logic LOW
2. The master then sends the address of the register on the slave that we want to read from, along with an extra bit to indicate that we want to read from this register (rather than writing to it)
    a. The memory of the slave is mapped into chunks called 'registers'. Different data are stored in different registers (e.g. the BMP388 stores pressure data across three different registers called *DATA_0*, *DATA_1*, and *DATA_2*), and each of these registers has an 'address' *(0x04*, *0x05*, and *0x06* in this case*)* that we can use to reference it. To read from one of these registers, we have to send it's address to the slave, so that the slave knows which register we want to access.
    b. The slave also has to know whether we want to read the data that is in the register, or write some data to the register. To indicate this, we add an extra '1' if we want to read, or '0' if we want to write. **Note: different devices will expect this extra 'read/write bit' in different locations (either before or after the register address).**
3. Now, the slave is ready to send data, so the master just sends an 'empty' signal onto the bus so that it can listen to the data that the slave is sending. The slave will send data 1 byte at a time – if we want to read several bytes, we just keep sending empty signals until we have what we need.
4. Now set the relevant SS pin back to logic HIGH to end the communications with the device.

Arduino implementation

On the next slide I will include a simple SPI implementation, just to show you some of the basic commands, and to better illustrate the communication process.

I will be trying to communicate with the *CHIP_ID* register on the BMP388. This is a good first step, as it allows us to read a register that contains a known value.

When we're setting up SPI, we should communicate with an identification register and check that the value we read is as we expect it to be. This will verify that we have successfully communicated with the device.

```
#include <SPI.h> /* include library for SPI commands (https://www.arduino.cc/en/reference/SPI) */
SPISettings SPIParams(10000000, MSBFIRST, SPI_MODE0); /* configure the  SPI to use a 10MHz clock signal, to send and receive data with the most significant bit first, and to use mode 0. */
const int altimeter_SS = 4; /* the altimeter slave select (SS) pin is connected to pin 4 on the Arduino in this example */

void setup() {
			unsigned int altimeter_CHIP_ID; /* declare a variable to store the value we read from the ID register */
			pinMode(altimeter_SS, OUTPUT); /* configure the pin that we have connected to the altimeters SS pin as an output */
			digitalWrite(altimeter_SS, HIGH); /* and then set it to high (this means we are not communicating with the device currently */
			SPI.begin(); /* initialise all the SPI lines and CPU to use SPI */
}

void loop() {
			byte registerSelect = 0x00 | (1 << 8); /* the address of the ‘CHIP_ID’ register is ‘0x00’. The altimeter expects the R/W bit to be the MSB of this command */
			SPI.beginTransaction(SPIParams); /* begin a transaction over SPI using our params. this command also stops interrupts from preventing SPI comms */
			digitalWrite(altimeter_SS, LOW); /* to communicate with the altimeter, we take its slave select pin on the PDC low */
			SPI.transfer(registerSelect); /* if we want to read a particular address, we must send the address of the register (& R/W bit) to the device */
			/* now if we send nothing, we are listening for the result - the device will send the value in the register we requested for the first byte, just read the value into 'result' */
			result = SPI.transfer(0x00);
			digitalWrite(deviceSelect, HIGH); /* stop communications with the altimeter by setting the corresponding slave select on the PDC to high */

			SPI.endTransaction(); /* we're done now! restart interrupt mechanisms */
			if (altimeter_CHIP_ID == 0x50) { return 0; } /* here you would implement some sort of check that things are as they should be ! */
}
```

Extra resources

I hope this was an easy to follow introduction! From here the concepts are pretty easy to apply to writing to registers or reading from any register of any device on the bus! Of course, I’m always happy to help should you need it!

Some online resources that might be useful:

All about circuits SPI intro:

https://www.allaboutcircuits.com/technical-articles/spi-serial-peripheral-interface/

Arduino SPI reference:

https://www.arduino.cc/en/reference/SPI