1: Communication between a Raspberry Pi and an Arduino (UART)

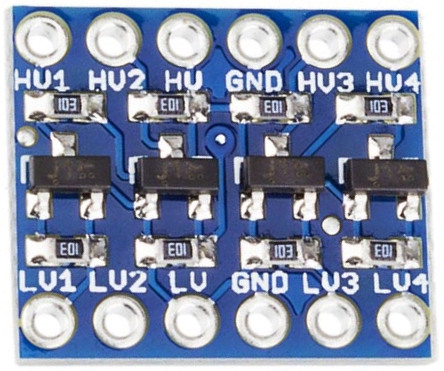
Let’s now see a different use case.

Being a keyboard rockstar was surely fun, but if I’m using Android Things to create a smart device, I don’t want to plug it on my computer over USB all the time! Can’t we find a better usage of UART?

Imagine that you are building a smart robot. At some point, you’ll find out that the Raspberry Pi has not enough pins to make all your dreams come true (*only 2 PWMs may not be enough for big needs*).  
How about delegating some stuff to a separate nano computer? In this use case, I’ll choose an Arduino.

Here comes the Bi-Directional Logic Level Converter

I’m using an Arduino here, instead of 2 Raspberry Pi to show you something tricky: The Raspberry Pi pins use 3.3V, while the Arduino pins use 5V.  
Connecting a Raspberry Pi pin to a voltage higher than 3.3V will likely damage your board.  
To plug these 2 components together, we will use a small chip that will adapt the voltage: a Bi-Directional Logic Level Converter.



HV on the chip means *“****H****igh****V****oltage”*, and LV *“****L****ow****V****oltage”*.  
Here, a low-voltage signal sent to LV will be shifted up to the higher voltage and sent out HV  
HV/LV 1 to 4 are for data *(GPIO/I2C/UART)*. HV/LV is for the voltage pin, And once again, GND, for the ground pin.

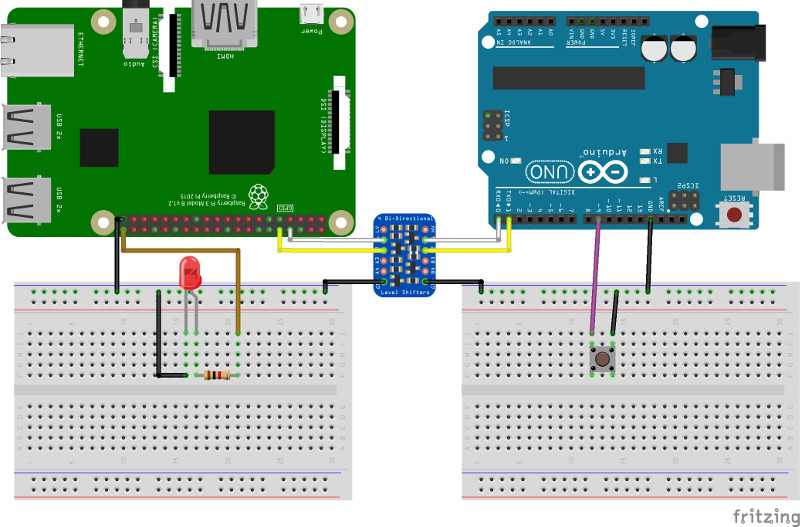
Since the Raspberry Pi’s using 3.3V it will be plugged to the LV side. The Arduino (5V) will be on the HV side.

Blinking LEDs, once again

Making an LED blink is our new *Hello, World!*

I will connect a button to the Arduino, and an LED to the Android Things Raspberry Pi.  
When the button is pressed, the Arduino will delegate its work to the Raspberry Pi via UART, so that the latter can turn its LED on/off.

Schematic



The only component linking the 2 nano-computers is the UART connection (Rx/Tx/Ground) passing through a Bi-Directional Logic Level Converter.

On the Android Things project, when receiving data, I will simply toggle the LED using the GPIO API:

**private** UartDeviceCallback mCallback **=** **new** UartDeviceCallback**()** **{**

@Override

**public** **boolean** **onUartDeviceDataAvailable(**UartDevice uart**)** **{**

ledGpio**.**setValue**(!**ledGpio**.**getValue**());**

**return** **true;**

**}**

And here’s the Arduino sketch:  
When pressing the button on pin #9, I send the “Pressed!” payload to the UART:

**int** buttonPin **=** 9;

**void** **setup**() {

pinMode(buttonPin, INPUT\_PULLUP);

Serial.begin(115200);

}

**void** **loop**() {

**if** (digitalRead(buttonPin) **==** LOW) {

Serial.println("Pressed!");

delay(1000);

}

}

# 2)raspberry pi to arduino spi communication

There are many reasons to include a dedicated microcontroller when developing hardware projects based on a Linux single board computer like the Raspberry Pi. Whatever the specifics, typically the end goal is to take advantage of the microcontroller's built-in peripherals such as analog inputs and PWM outputs. Additionally, microcontrollers provide multiple timers, interrupts and other features to make easy work of precision real time processing and control.

With its enduring popularity, it's no surprise that an Arduino variant is often the go-to solution for such projects. The platform provides a robust, general purpose microcontroller appropriate for many projects, with a rich ecosystem of libraries, tutorials, and other resources.

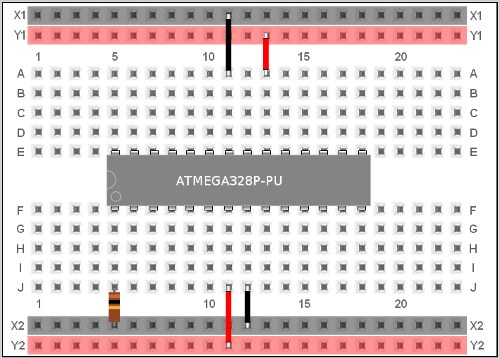
This tutorial presents a basic framework for Raspberry Pi to Arduino communication and control using SPI - the Serial Peripheral Interface bus. SPI represents a very well established chip-to-chip communication methodology that is implemented in hardware on both devices. Here we will explore SPI in some detail, discuss hardware and software considerations, and develop a working example of a bi-directional communications scheme that could be adapted for any number of command and control applications.

## arduino at 3.3 volts

Choosing an Arduino that runs at 3.3v will simplify your Raspberry Pi Arduino projects by enabling a direct connection between the input and output pins on the two devices without the need for level-shifting to align voltages and protect the Raspi inputs.

Arduino is often described as a 5 volt platform, but this is not entirely true. The ATMEGA328P-PU microcontroller at the heart of many standard Arduinos is actually rated to operate from 1.8v to 5.5v. Although the flagship Arduino Uno and a number of other products are designed for 5v operation, there are also plenty of official and unofficial 3.3v Arduinos to choose from, including the Fio, Pro, Pro-Mini and Lilypad. (Pro and Pro-Mini have both 3.3v and 5v versions.)

The "Arduino" used in developing this tutorial is an ATMEGA328P-PU in a barebones configuration on a breadboard. The setup is truly barebones. There is no voltage regulator or external clock crystal, just the chip powered by the Raspi's 3.3v tap and a 10K resistor to pull the reset pin high. (Of course it wouldn't hurt to put a few decoupling capacitors on the power rails.) In addition to the enticing simplicity and frugality of the circuit, the barebones configuration is ideal for connecting to the Raspberry Pi. Not only is the 3.3v operation Raspi-friendly, but foregoing the voltage regulator and crystal significantly reduces overall power consumption which is especially important on older Pis which have a 50mA limit on the 3.3v line.



I routinely use this barebones setup on small, battery powered projects. When brewing your own, there are a few considerations to running the 328P at lower voltages. The most significant concern is the need to run at a lower clock speed - 8MHz is the fastest clock that can be reliably supported at 3.3v. Another consideration is that the 328's brownout detection must either be disabled or configured such that the dropout is less than 3.3v.

The official Arduino site has a board.txt file here: [From Arduino to a Microcontroller on a Breadboard](http://www.arduino.cc/en/Tutorial/ArduinoToBreadboard) that will provide the proper settings to run a minimalist board and load an 8MHz compatible bootloader. If you're new to burning bootloaders and building your own boards, this page will also provide a nice jumping off point to gathering the necessary knowledge and skills.

## a few notes on the raspberry pi environment and c++ compiler

This tutorial was developed and tested using both Raspberry Pi Model B and Raspberry Pi 2 running the standard Raspian Linux distribution. The examples should work without modification on any current Raspberry Pi model. I run my Raspi "headless" (without monitor or keyboard) connecting from a terminal on a Linux PC via SSH, so virtually all of my interaction is at the command line. The code for the examples is written in C++ and compiled using the GNU Compiler Collection (GCC). GCC is part of the Raspian distro and should therefore be available without the need for installation or configuration. All of the examples here can be compiled using a very simple command line syntax:

*g++ -o <target> <source>*

Super user permissions are required for access to the SPI peripheral, so compiled examples will need to be executed using sudo, as in:

*sudo ./example*

## the spi interface

SPI is one of several viable options for Raspberry Pi Arduino connectivity. A few advantages of the SPI interface are widespread adoption, signal integrity, flexibility in protocol design, and simplicity.

SPI employs a master-slave architecture. The master device initiates and controls all communication with the slave. SPI communication is synchronous, meaning that data transmission between the two devices is synchronized by a shared clock signal. SPI communication is also full-duplex, meaning that data can flow in both directions simultaneously.

The master controls the data transmission via the clock line. Every time the master pulses the clock line, one bit of data is pushed from the master to the slave. Simultaneously one bit of data is pushed from the slave to the master. As a practical matter, and for our purposes here, data can be thought of as moving at the byte level. The master and slave each load a byte into their respective SPI data registers, and the master pulses the clock line eight times to push one byte in each direction. Although there are various hardware flags that can be set and interrupts that can triggered, there is no SPI protocol. The simultaneous transfer of one byte of data between the two devices is the entirety of the process. If required, acknowledgement signals, error checking, packet definition and other common protocol features can all be handled in software.

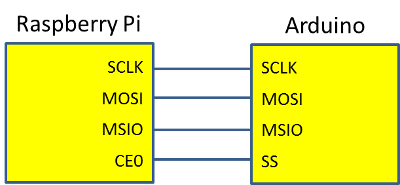
The SPI bus itself is made up of four physical connections: two data lines, a clock line, and the slave select line. The master must have one slave select line for each slave, so the number of required slave select lines could theoretically grow large.

Nomenclature and labeling for the clock and data lines are identical for the Raspberry Pi and Arduino:

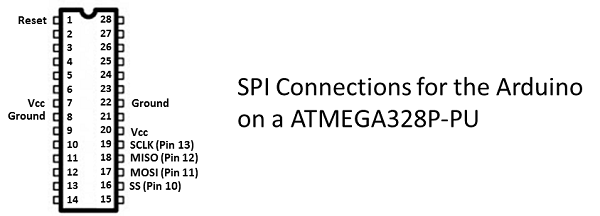
* SCLK : Serial Clock (output from master).
* MOSI : Master Output, Slave Input (output from master).
* MISO : Master Input, Slave Output (output from slave).

However on the slave select line, nomenclature is slightly different between the two platforms. On the Arduino the line is called Slave Select and is abbreviated SS. On the Raspberry Pi, the process is called Chip Select, abbreviated CS, and the CS lines are referred to as Chip Enable lines, abbreviated CE. The PI actually makes two slave select lines available identified as CE0 and CE1.

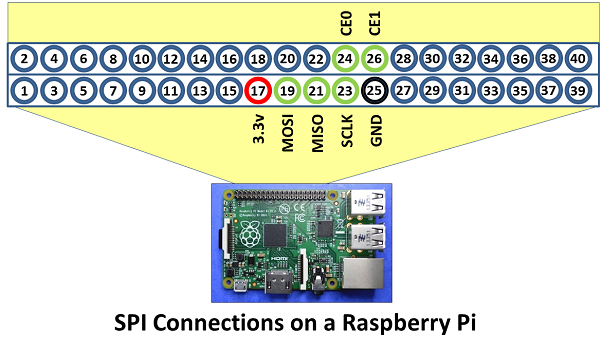
In contrast to some serial bus schemes where the transmission line (TX) from one device is connected to the receive line (RX) on the other, in SPI the four lines on one device are wired directly to their counterpart on the other as shown in the diagram below.



The physical location of the SPI bus on the ATMEGA328P-PU is straightforward as depicted in the pinout diagram below. SS is on physical pin 16, MOSI on pin 17, MISO on pin 18 and SCLK on pin 19. If you are using a manufactured board instead of a raw chip, these physical pins correspond to the Arduino digital pins 10 to 13.



A pinout diagram for the Raspberry Pi GPIO header is even more straightforward. Following the pinout diagram below (and consistent among version A, A+, B, B+, and 2B) MOSI is pin 19, MISO pin 21, and SCLK pin 23. In this tutorial we're using CS CE0 which is on pin 24.



## enabling spi on the raspberry pi

SPI on the Pi is disabled by default. Before you can use SPI in your programs it must be enabled. The easiest way is via the raspi-config utility which you can launch by entering:

*sudo raspi-config*

at the command line.

In the resulting screens select Advanced Options and then select SPI. You should answer yes to both questions to enable the SPI interface and load the SPI kernel module by default.

Reboot the Pi to load the SPI kernel module. You can verify that the change was effective by entering *lsmod* at the command line to generate a listing of loaded modules. The SPI module *spi\_bcm2708* should appear in the results.

This should all work fairly effortlessly, however, if the SPI module fails to load you may find yourself needing to troubleshoot the activation. It is possible to manually examine the Pi's configuration files to ensure the settings are correct using nano or the text editor of your choice. Note that if you need to make changes you will need to be running the text editor as sudo.

The Pi has a "blacklist" file which blocks modules from being loaded: */etc/modprobe.d/raspi-blacklist.conf*. The line *#blacklist spi-bcm2708* must begin with the hash symbol "#" which "comments out" the line. Without the hash, SPI is blacklisted and won't load.

Another potential culprit is the Device Tree parameter settings in the */boot/config.txt* file. Look for a line in the file which reads *dtparam=spi=on*. If the line reads that spi is off, or the line is missing altogether, then edit the file accordingly. It is fine to simply make the line the last line in the file.

Remember that you will need to reboot the Pi before any kernel module changes will take effect.

## coding spi on the raspberry pi

Raspian Linux provides several layers of abstraction to the SPI bus. To gain access to the bus, a virtual file is opened that points to the SPI device, and a file descriptor is assigned. Subsequent interactions with the SPI device are made by reading and writing to that file. The SPI virtual file will follow the naming convention of */dev/spidevB.C* where B is the device number and C is the chip select. In our case, device 0 and chip select 0, we will open the file */dev/spidev0.0* for read/write access, setting the variable *fd* equal to the file descriptor:

fd **=** open**(**"/dev/spidev0.0"**,** O\_RDWR**);**

With the file now open, communication with the SPI hardware is made via the ioctl (input/output control) system call. The spidev device driver provides an interface for passing information to and from the SPI hardware via ioctl. The next step before actually transmitting and receiving data is to set configuration options for the session. The spidev driver provides functionality to set/override various SPI options that include the SPI Mode for clock polarity and clock phase, the bit width of transmitted words, the bit order of transmission, and the maximum speed of transmission. For the most part, the Raspberry Pi Arduino communications project uses the Raspberry Pi's default settings: SPI Mode 0, 8 bits per word, and most significant bit first. (For clarity's sake, these are actually the default settings of the Broadcom SPI peripheral hardware itself.)

One setting worth considering is the maximum transmission speed. The Atmel reference for SPI states that the system clock of the slave must be at least 4 times the SPI transmission rate. Since the barebones ATMEGA328P we're using is configured for an 8MHz clock, transmission from the Pi master could be as high as 2MHz. Here we'll set the SPI clock speed to 1MHz with a system call to ioctl, using the spidev defined SPI\_IOC\_WR\_MAX\_SPEED\_HZ and the value 1000000. Note that all values must be passed to ioctl by reference. First we assign the desired value to a variable (in this case *speed* and then rather than pass the actual variable, we pass a pointer to the variable using the syntax *&speed*.

**unsigned** **int** speed **=** **1000000;**

ioctl **(**fd**,** SPI\_IOC\_WR\_MAX\_SPEED\_HZ**,** **&**speed**);**

The SPI device is now ready to transmit and receive data. The spidev driver establishes a data structure *spi\_ioc\_transfer* that is used to format both data and any temporary configuration data to be passed to the device via ioctl. Here we'll use that structure and create a variable named *spi* that can be passed by reference to ioctl. We'll also use memset to allocate memory for the struct:

**struct** spi\_ioc\_transfer spi**;**

memset **(&**spi**,** **0,** **sizeof** **(**spi**))** **;**

Again, we're mostly content with the default settings for the transfer, so we'll only need to load the struct with three pieces of information:

* .tx\_buf - a pointer to a variable holding the data to be transferred
* .rx\_buf - a pointer to a variable to hold received data
* .len - length in bytes of tx and rx buffers

Two variables will need to be initiated for the transfer and receive buffers, which we'll name *txDat* and *rxDat*. Although spidev will enable multibyte transmissions, the Raspberry Pi Arduino project will only use single byte transfers, so the length value will be set at 1.

**unsigned** **char** txDat**;**

**unsigned** **char** rxDat**;**

**...**

spi**.**tx\_buf **=** **(unsigned** **long)&**txDat**;**

spi**.**rx\_buf **=** **(unsigned** **long)&**rxDat**;**

spi**.**len **=** **1;**

All that remains is to call ioctl and pass the spi struct to the virtual file:

ioctl **(**fd**,** SPI\_IOC\_MESSAGE**(1),** **&**spi**);**

The txDat byte will be transmitted to the Arduino and the returned byte will be in rxDat.

Putting the pieces together, the following function, *spiTxRx*, accepts a single byte, sends that byte by SPI, and returns the received byte:

**int** spiTxRx**(unsigned** **char** txDat**)**

**{**

**unsigned** **char** rxDat**;**

**struct** spi\_ioc\_transfer spi**;**

memset **(&**spi**,** **0,** **sizeof** **(**spi**));**

spi**.**tx\_buf **=** **(unsigned** **long)&**txDat**;**

spi**.**rx\_buf **=** **(unsigned** **long)&**rxDat**;**

spi**.**len **=** **1;**

ioctl **(**fd**,** SPI\_IOC\_MESSAGE**(1),** **&**spi**);**

**return** rxDat**;**

**}**

Lastly, to enable the functionality described above, the following header files must be referenced for inclusion in your C++ source file:

*#include <sys/ioctl.h>*

*#include <linux/spi/spidev.h>*

*#include <fcntl.h>*

*#include <cstring>*

*#include <iostream>*

## spi on the arduino

The ATMEGA328P is equipped with a fully configurable high speed SPI peripheral that can function as either SPI master or slave. Interaction with the bus is conducted via three hardware registers - the SPI Control Register (SPCR), the SPI Status Register (SPSR) and the SPI Data Register (SPDR).

We will be configuring the device as a slave. As with the earlier configuration of the Raspberry Pi, we are able to rely mostly on the default settings of the ATMEGA328P, so there will be no need to make changes to settings for mode or bit order. There is also no need to configure clock speed on a slave device because the master drives the clock. While an Arduino SPI library does exist, it does not have support for SPI slave operation. Fortunately, the desired functionality can be easily established by direct register access. In fact, it only takes two lines of code to setup the SPI slave.

First the MISO pin has to be defined as an output pin. All other pins are configured automatically as input pins if the SPI is enabled:

pinMode**(**MISO**,** OUTPUT**);**

Second the SPI enable bit needs to be set:

SPCR **|=** \_BV**(**SPE**);**

Reading and writing of SPI data is performed through SPDR. Programmatically you can treat SPDR as you would a variable. To read the contents of SDPR, it can either be accessed directly, or another variable can be set equal to it:

i **=** SPDR**;**

To load the data register with a value to transmit back to the master, the statement is reversed:

SPDR **=** i**;**

At the hardware level SPDR includes both an 8-bit shift register and an 8-bit receive buffer. When the slave is receiving data, that data is shifted into the shift register one bit at a time while the original 8-bits in the register are shifted back to the master. When a complete byte has been shifted into the register, that byte is then copied into the receive buffer. The receive buffer won't be updated again until the next complete byte is received.

There are several implications of this sequence of operations. A byte to be transmitted cannot be written to SPDR until the current transmission cycle is completed, but must be in SPDR before the next transmission cycle begins. The byte in the receive buffer must be read before the next transmission cycle is completed or it will be overwritten. However, reading SPDR is actually reading the receive buffer and writing to SPDR does not overwrite the receive buffer. Therefore it is possible to write the next byte prior to reading the received byte.

Because the master device will initiate and control communications by shifting a byte into the slave device, the slave requires a method for recognizing that a byte has been received. There are two possible methods for an Arduino/ATMEGA slave to monitor the SPI device - interrupt or polled. In an interrupt driven scheme, the interrupt enable bit in SPCR is set, and when a byte is received the Arduino immediately stops processing and executes a special function called an Interrupt Service Routine (ISR). In a polled scheme, the Arduino is coded to periodically check a flag in SPSR which will indicate that a byte has been received.

The choice to use interrupts or polling is based on the specifics of application being developed. Polling places a greater burden on software to continually monitor SPSR for a change. Interrupts create extra overhead to enter and exit the ISR, and extra care must be taken to avoid conflicts with other interrupt and timer dependent processes. In the code developed for this tutorial, polling is used.

The SPI End of Transmission Flag (SPIF) in SPSR is set when a byte has been received. Reading or writing SPDR clears SPIF. A quick method for checking the status of a single register bit is to mask off all the other bits in the register with an AND operation and then test whether or not the result is zero. For the tutorial, we'll use an IF-THEN construct to test for the flag and execute the code within the IF statement anytime the flag is set. If the executed code includes a read or write of SPDR the SPIF flag will automatically be cleared:

**if((**SPSR **&** **(1** **<<** SPIF**))** **!=** **0)**

**{**

dat **=** SPDR**;**

**}**

## hello raspi

The *SPI\_Hello\_Raspi* and *SPI\_Hello\_Arduino* programs listed below, provide a demonstration of a minimal working Raspberry Pi Arduino communications system. In addition to putting some shape to the concepts introduced thus far, the demo can be invaluable in establishing that you have a working SPI connection, and that the correct kernel module and drivers are available and working on the Pi.

Otherwise, as with any "Hello World" program, the setup accomplishes little. On the Pi, a character array is established, and the ASCII characters for Hello Arduino are pushed through the SPI bus one byte at a time. On the Arduino the characters for Hello Raspi are stored in a similar array. As bytes are received on the Arduino, SPDR is loaded with the Hello Raspi characters and, driven by the clock from the Pi master, they are sent to the Pi one byte at a time. This sequence is in a loop that will repeat endlessly until the program on the Pi is forced to break. (Control Z will do the trick.)

The primary element to observe in the example is the coordination between the devices. The Raspberry Pi sends its byte and then pauses for 10 microseconds. The Arduino needs a small window of time in order to load SPDR with next the byte to send to the Raspi. The Arduino can't load the byte until the Raspi completes the current one-byte transmission, but the Arduino must have loaded the byte before the Raspi begins sending the next byte. On the Arduino side, the code loops endlessly until SPIF is set indicating the byte has been received and SPDR can be loaded.

Lastly, the astute will notice that the Arduino is not processing the incoming data. However, because of the master/slave architecture it is necessary to have the Raspi push one byte of data through for each byte it receives, so no harm was done saying hello to the Arduino. (In a less frivolous application, one might push through a series of zeros to accomplish the same result.)

## hello arduino (raspberry pi code)

*/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\**

*SPI\_Hello\_Arduino*

*Configures an Raspberry Pi as an SPI master and*

*demonstrates bidirectional communication with an*

*Arduino Slave by repeatedly sending the text*

*"Hello Arduino" and receiving a response*

*Compile String:*

*g++ -o SPI\_Hello\_Arduino SPI\_Hello\_Arduino.cpp*

*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/*

*#include <sys/ioctl.h>*

*#include <linux/spi/spidev.h>*

*#include <fcntl.h>*

*#include <cstring>*

*#include <iostream>*

**using** **namespace** std**;**

*/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\**

*Declare Global Variables*

*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/*

**int** fd**;**

**unsigned** **char** hello**[]** **=** **{**'H'**,**'e'**,**'l'**,**'l'**,**'o'**,**' '**,**

'A'**,**'r'**,**'d'**,**'u'**,**'i'**,**'n'**,**'o'**};**

**unsigned** **char** result**;**

*/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\**

*Declare Functions*

*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/*

**int** spiTxRx**(unsigned** **char** txDat**);**

*/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\**

*Main*

*Setup SPI*

*Open file spidev0.0 (chip enable 0) for read/write*

*access with the file descriptor "fd"*

*Configure transfer speed (1MkHz)*

*Start an endless loop that repeatedly sends the characters*

*in the hello[] array to the Ardiuno and displays*

*the returned bytes*

*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/*

**int** main **(void)**

**{**

fd **=** open**(**"/dev/spidev0.0"**,** O\_RDWR**);**

**unsigned** **int** speed **=** **1000000;**

ioctl **(**fd**,** SPI\_IOC\_WR\_MAX\_SPEED\_HZ**,** **&**speed**);**

**while** **(1)**

**{**

**for** **(int** i **=** **0;** i **<** **sizeof(**hello**);** i**++)**

**{**

result **=** spiTxRx**(**hello**[**i**]);**

cout **<<** result**;**

usleep **(10);**

**}**

**}**

**}**

*/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\**

*spiTxRx*

*Transmits one byte via the SPI device, and returns one byte*

*as the result.*

*Establishes a data structure, spi\_ioc\_transfer as defined*

*by spidev.h and loads the various members to pass the data*

*and configuration parameters to the SPI device via IOCTL*

*Local variables txDat and rxDat are defined and passed by*

*reference.*

*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/*

**int** spiTxRx**(unsigned** **char** txDat**)**

**{**

**unsigned** **char** rxDat**;**

**struct** spi\_ioc\_transfer spi**;**

memset **(&**spi**,** **0,** **sizeof** **(**spi**));**

spi**.**tx\_buf **=** **(unsigned** **long)&**txDat**;**

spi**.**rx\_buf **=** **(unsigned** **long)&**rxDat**;**

spi**.**len **=** **1;**

ioctl **(**fd**,** SPI\_IOC\_MESSAGE**(1),** **&**spi**);**

**return** rxDat**;**

**}**

## hello raspi (arduino code)

*/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\**

*SPI\_Hello\_Raspi*

*Configures an ATMEGA as an SPI slave and demonstrates*

*bidirectional communication with an Raspberry Pi SPI master*

*by repeatedly sending the text "Hello Raspi"*

*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/*

*/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\**

*Global Variables*

*-hello[] is an array to hold the data to be transmitted*

*-marker is used as a pointer in traversing data arrays*

*/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/*

**unsigned** **char** hello**[]** **=** **{**'H'**,**'e'**,**'l'**,**'l'**,**'o'**,**' '**,**

'R'**,**'a'**,**'s'**,**'p'**,**'i'**,**'\n'**};**

byte marker **=** **0;**

*/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\**

*Setup SPI in slave mode (1) define MISO pin as output (2) set*

*enable bit of the SPI configuration register*

*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/*

**void** setup **(void)**

**{**

pinMode**(**MISO**,** OUTPUT**);**

SPCR **|=** \_BV**(**SPE**);**

**}**

*/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\**

*Loop until the SPI End of Transmission Flag (SPIF) is set*

*indicating a byte has been received. When a byte is*

*received, load the next byte in the Hello[] array into SPDR*

*to be transmitted to the Raspberry Pi, and increment the marker.*

*If the end of the Hell0[] array has been reached, reset*

*marker to 0.*

*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/*

**void** loop **(void)**

**{**

**if((**SPSR **&** **(1** **<<** SPIF**))** **!=** **0)**

**{**

SPDR **=** hello**[**marker**];**

marker**++;**

**if(**marker **>** **sizeof(**hello**))**

**{**

marker **=** **0;**

**}**

**}**

**}**

## a more robust example

To provide a more robust example of Raspberry Pi Arduino communication we'll now present a simple application in which the Pi sends a command to the Arduino to add or subtract two numbers and the Arduino returns the result. Creating an application that solves a math problem avoids the need to interface any additional hardware or other circuitry while providing a fairly straightforward troubleshooting path. Nonetheless, the application has many fundamental elements of a master slave system and it should be easy to appreciate how the techniques could be applied to a more realistic scenario such as reading an analog sensor.

As seen in the Hello Raspi application, accomplishing even the simplest of exchanges between devices requires establishing and adhering to a rigid set of expectations regarding the timing and sequence of communications and other events - in other words a communications protocol. We've actually already made quite a bit of progress toward developing a basic, general purpose protocol that can be expanded or adapted for specific purposes.

By accepting certain default settings of the Raspberry Pi and Arduino, we've established that the evolving protocol uses SPI Mode 0, transmission words are 8 bits, and the most significant bit is transmitted first. Additionally we've determined that transmission rate is 1MHz with a 10 microsecond delay between bytes.

To support our math application we'll be adding just a little more structure. Specifically, we'll add a handshaking step to the beginning of each transmission, and we'll also establish a fixed packet size for transmissions.

## handshaking

The handshaking step to be added is as simple as they come. The Raspberry Pi will send a one-byte start code ("c") which the Arduino will acknowledge with a one-byte acknowledge code ("a"). Both devices will loop until the sequence has been successfully executed. As simple as this technique may be, it effectively addresses two potential problems that are likely to crop up as soon as you begin incorporating SPI communications into actual projects.

The first issue being addressed is that using only the SPI bus, an SPI master has no way of confirming that the slave is actually online and receiving information. If the Arduino were offline or otherwise unavailable, the Raspberry Pi would cheerfully send a stream of data into the ether. The Pi will also quite happily read the incoming data as a string of zeros or whatever random noise may be present on the MISO line.

The other issue addressed by handshaking is the potential for the slave to either miss a byte somewhere along the way or start in the middle of a transmission. Should either scenario occur without a mechanism to realign the master and slave, transmission from that point onwards would be somewhat hopelessly out of synch. With the handshaking sequence, the two devices should realign on the very next transmission.

## packets

In most situations communication between the master and slave device will involve a multi-byte packet. The optimal size of the packet depends very much on the application being developed. If the application is sending large amounts of raw data, a larger packet size might be in order. If the application is sending short command sequences, then a much smaller packet might be called for. Likewise, the decision on whether to use fixed or variable packet length throughout the application will depend on the nature of the application.

If you look to the Raspberry Pi source code below you'll see that the command to either add or subtract two numbers is called in the following format:

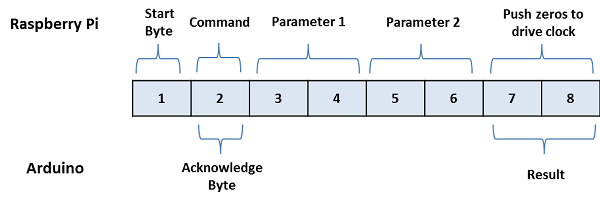
results **=** sendCommand**(**command**,** parameter1**,** parameter2**);**

Where *sendCommand* is a function, *command* is a single byte instruction, and *parameter1*, *parameter2*, and *results* are all two-byte integers. When called, the function packages the command and parameters, and transmits them to the Arduino by SPI. The Arduino performs the calculation and transmits the results back to the Pi, where *sendCommand* will return the Arduino's response as the result of the function. The program uses the letter 'a' as the command for addition and the letter 's' as the command for subtraction. A command to add two numbers appears as:

results **=** sendCommand**(**'a'**,** **510,** **655);**

Either command is the same in terms of length, therefore the optimal packet size is simply the number of bytes required to conduct the handshake, send the command and two integer parameters, and return the result: 8 bytes.

The graphic below depicts the packet and can serve as a guide to understanding the sequence of events:



## of bytes and integers

In transmitting data between the Raspberry Pi and Arduino we encounter a very common problem in serial communication. The SPI unit of transmission is a byte, but the parameters and result required for our application are two-byte integers. In order to work fluidly we must have a method of splitting the integers into two bytes for transmission and then reassembling the bytes into an integer once they are received. Bit shifting is one standard method of dealing with this conversion requirement. However, here we'll use a perhaps more obscure, but nonetheless elegant solution. C++ has a construct called *union* which allows two or more variables to occupy the same memory space. In the snippet below a union named p1Buffer contains both the integer variable p1Int and a two byte array variable p1Char[2]:

**union**

**{**

**int** p1Int**;**

**unsigned** **char** p1Char**[2];**

**}** p1Buffer**;**

Later in the program we are able to seamlessly refer to the contents of the union by either referring to the integer - *p1Buffer.p1Int* or either of the bytes - *p1Buffer.p1Char[0]* or *p1Buffer.p1Char[1]*.

## the code

With the above description, the code below should be fairly straightforward to navigate.

On the Raspberry Pi, the *sendCommand* function serves as a wrapper for the *spiTxRx* function we introduced in the Hello Arduino application and executes our protocol step by step.

On the Arduino, we've added a function called *spiHandler* This function is called every time the SPIF is set and uses a variable named marker to keep track of its position within the packet. The switch case construct relies on the value of marker to be able to execute the appropriate action at each step of the protocol, from responding to the handshake request, to interpreting the command and parameters, to return the result.

As before the program itself executes in an endless loop performing first an addition problem and then a subtraction problem, and then sending the results to the console.

## spi raspi arduino (raspberry pi code)

*/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\**

*SPI\_Raspi\_Arduino*

*Configures an Raspberry Pi as an SPI master and*

*demonstrates a basic bidirectional communication scheme*

*with an Arduino slave. The Raspberry Pi transmits*

*commands to perform addition and subtraction on a pair*

*of integers and the Ardunio returns the result*

*Compile String:*

*g++ -o SPI\_Raspi\_Arduino SPI\_Raspi\_Arduino.cpp*

*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/*

*#include <sys/ioctl.h>*

*#include <linux/spi/spidev.h>*

*#include <fcntl.h>*

*#include <iostream>*

*#include <cstring>*

**using** **namespace** std**;**

*/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\**

*Housekeeping variables*

*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/*

**int** results**;**

**int** fd**;**

*/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\**

*Declare Functions*

*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/*

**int** spiTxRx**(unsigned** **char** txDat**);**

**int** sendCommand**(char** i**,** **int** j**,** **int** k**);**

*/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\**

*Main*

*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/*

**int** main **(void)**

**{**

*/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\**

*Setup SPI*

*Open file spidev0.0 (chip enable 0) for read/write access*

*with the file descriptor "fd"*

*Configure transfer speed (1MkHz)*

*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/*

fd **=** open**(**"/dev/spidev0.0"**,** O\_RDWR**);**

**unsigned** **int** speed **=** **1000000;**

ioctl **(**fd**,** SPI\_IOC\_WR\_MAX\_SPEED\_HZ**,** **&**speed**);**

*/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\**

*An endless loop that repeatedly sends the demonstration*

*commands to the Arduino and displays the results*

*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/*

**while** **(1)**

**{**

results **=** sendCommand**(**'a'**,** **510,** **655);**

cout **<<** "Addition results:" **<<** endl**;**

cout **<<** "510 + 655 = " **<<** **(int)(**results**)** **<<** endl**;**

results **=** sendCommand**(**'s'**,** **1000,** **250);**

cout **<<** "Subtraction results:" **<<** endl**;**

cout **<<** "1000 - 250 = " **<<** **(int)(**results**)** **<<** endl **<<**endl**;**

sleep**(1);**

**}**

**}**

*/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\**

*spiTxRx*

*Transmits one byte via the SPI device, and returns one byte*

*as the result.*

*Establishes a data structure, spi\_ioc\_transfer as defined*

*by spidev.h and loads the various members to pass the data*

*and configuration parameters to the SPI device via IOCTL*

*Local variables txDat and rxDat are defined and passed by*

*reference.*

*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/*

**int** spiTxRx**(unsigned** **char** txDat**)**

**{**

**unsigned** **char** rxDat**;**

**struct** spi\_ioc\_transfer spi**;**

memset **(&**spi**,** **0,** **sizeof** **(**spi**));**

spi**.**tx\_buf **=** **(unsigned** **long)&**txDat**;**

spi**.**rx\_buf **=** **(unsigned** **long)&**rxDat**;**

spi**.**len **=** **1;**

ioctl **(**fd**,** SPI\_IOC\_MESSAGE**(1),** **&**spi**);**

**return** rxDat**;**

**}**

*/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\**

*sendCommand*

*Demonstration of a protocol that uses the spiTxRx function*

*to send a formatted command sequence/packet to the Arduino*

*one byte at and capture the results*

*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/*

**int** sendCommand**(char** command**,** **int** j**,** **int** k**)**

**{**

**unsigned** **char** resultByte**;**

**bool** ack**;**

*/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\**

*Unions allow variables to occupy the same memory space*

*a convenient way to move back and forth between 8-bit and*

*16-bit values etc.*

*Here three unions are declared: two for parameters to be*

*passed in commands to the Arduino and one to receive*

*the results*

*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/*

**union** p1Buffer\_T

**{**

**int** p1Int**;**

**unsigned** **char** p1Char **[2];**

**}** p1Buffer**;**

**union** p2Buffer\_T

**{**

**int** p2Int**;**

**unsigned** **char** p2Char **[2];**

**}** p2Buffer**;**

**union** resultBuffer\_T

**{**

**int** resultInt**;**

**unsigned** **char** resultChar **[2];**

**}** resultBuffer**;**

p1Buffer**.**p1Int **=** j**;**

p2Buffer**.**p2Int **=** k**;**

resultBuffer**.**resultInt **=** **0;**

*/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\**

*An initial handshake sequence sends a one byte start code*

*('c') and loops endlessly until it receives the one byte*

*acknowledgment code ('a') and sets the ack flag to true.*

*(Note that the loop also sends the command byte while*

*still in handshake sequence to avoid wasting a transmit*

*cycle.)*

*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/*

**do**

**{**

ack **=** false**;**

spiTxRx**(**'c'**);**

usleep **(10);**

resultByte **=** spiTxRx**(**command**);**

**if** **(**resultByte **==** 'a'**)**

**{**

ack **=** true**;**

**}**

usleep **(10);**

**}**

**while** **(**ack **==** false**);**

*/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\**

*Send the parameters one byte at a time.*

*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/*

spiTxRx**(**p1Buffer**.**p1Char**[0]);**

usleep **(10);**

spiTxRx**(**p1Buffer**.**p1Char**[1]);**

usleep **(10);**

spiTxRx**(**p2Buffer**.**p2Char**[0]);**

usleep **(10);**

spiTxRx**(**p2Buffer**.**p2Char**[1]);**

usleep **(10);**

*/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\**

*Push two more zeros through so the Arduino can return the*

*results*

*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/*

resultByte **=** spiTxRx**(0);**

resultBuffer**.**resultChar**[0]** **=** resultByte**;**

usleep **(10);**

resultByte **=** spiTxRx**(0);**

resultBuffer**.**resultChar**[1]** **=** resultByte**;**

**return** resultBuffer**.**resultInt**;**

**}**

## spi raspi arduino (arduino code)

*/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\**

*SPI\_Raspi\_Arduino*

*Configures an ATMEGA as an SPI slave and demonstrates*

*a basic bidirectional communication scheme*

*A Raspberry Pi SPI master transmits commands to*

*perform addition and subtraction on a pair of integers and*

*the Ardunio transmits the result*

*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/*

*/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\**

*Global Variables*

*-receiveBuffer[] and dat are used to capture incoming data*

*from the Raspberry Pi*

*-marker is used as a pointer to keep track of the current*

*position in the incoming data packet*

*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/*

**unsigned** **char** receiveBuffer**[5];**

**unsigned** **char** dat**;**

byte marker **=** **0;**

*/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\**

*Unions allow variables to occupy the same memory space a*

*convenient way to move back and forth between 8-bit and*

*16-bit values etc. Here three unions are declared:*

*two for parameters that are passed in commands to the Arduino*

*and one to receive the results*

*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/*

**union**

**{**

**int** p1Int**;**

**unsigned** **char** p1Char **[2];**

**}** p1Buffer**;**

**union**

**{**

**int** p2Int**;**

**unsigned** **char** p2Char **[2];**

**}** p2Buffer**;**

**union**

**{**

**int** resultInt**;**

**unsigned** **char** resultChar **[2];**

**}** resultBuffer**;**

*/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\**

*Setup SPI in slave mode (1) define MISO pin as output (2) set*

*enable bit of the SPI configuration register*

*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/*

**void** setup **(void)**

**{**

pinMode**(**MISO**,** OUTPUT**);**

SPCR **|=** \_BV**(**SPE**);**

**}**

*/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\**

*Loop until the SPI End of Transmission Flag (SPIF) is set*

*indicating a byte has been received. When a byte is*

*received, call the spiHandler function.*

*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/*

**void** loop **(void)**

**{**

**if((**SPSR **&** **(1** **<<** SPIF**))** **!=** **0)**

**{**

spiHandler**();**

**}**

**}**

*/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\**

*spiHandler*

*Uses the marker variable to keep track current position in the*

*incoming data packet and execute accordingly*

*0 - wait for to receive start byte - once received send*

*the acknowledge byte*

*1 - the command to add or subtract*

*2-5 - two integer parameters to be added or subtracted*

*- when the last byte (5) is received, call the*

*executeCommand function and load the first byte of the*

*result into SPDR*

*6 - transmit the first byte of the result and load the*

*second byte into SPDR*

*7 - transmit the second byte of of the result and reset*

*the marker*

*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/*

**void** spiHandler**()**

**{**

**switch** **(**marker**)**

**{**

**case** **0**:

dat **=** SPDR**;**

**if** **(**dat **==** 'c'**)**

**{**

SPDR **=** 'a'**;**

marker**++;**

**}**

**break;**

**case** **1**:

receiveBuffer**[**marker**-1]** **=** SPDR**;**

marker**++;**

**break;**

**case** **2**:

receiveBuffer**[**marker**-1]** **=** SPDR**;**

marker**++;**

**break;**

**case** **3**:

receiveBuffer**[**marker**-1]** **=** SPDR**;**

marker**++;**

**break;**

**case** **4**:

receiveBuffer**[**marker**-1]** **=** SPDR**;**

marker**++;**

**break;**

**case** **5**:

receiveBuffer**[**marker**-1]** **=** SPDR**;**

marker**++;**

executeCommand**();**

SPDR **=** resultBuffer**.**resultChar**[0];**

**break;**

**case** **6**:

marker**++;**

SPDR **=** resultBuffer**.**resultChar**[1];**

**break;**

**case** **7**:

dat **=** SPDR**;**

marker**=0;**

**}**

**}**

*/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\**

*executeCommand*

*When the complete 5 byte command sequence has been received*

*reconstitute the byte variables from the receiveBuffer*

*into integers, parse the command (add or subtract) and perform*

*the indicated operation - the result will be in resultBuffer*

*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/*

**void** executeCommand**(void)**

**{**

p1Buffer**.**p1Char**[0]=**receiveBuffer**[1];**

p1Buffer**.**p1Char**[1]=**receiveBuffer**[2];**

p2Buffer**.**p2Char**[0]=**receiveBuffer**[3];**

p2Buffer**.**p2Char**[1]=**receiveBuffer**[4];**

**if(**receiveBuffer**[0]** **==** 'a'**)**

**{**

resultBuffer**.**resultInt **=** p1Buffer**.**p1Int **+** p2Buffer**.**p2Int**;**

**}**

**else** **if** **(**receiveBuffer**[0]** **==** 's'**)**

**{**

resultBuffer**.**resultInt **=** p1Buffer**.**p1Int **-** p2Buffer**.**p2Int**;**

**}**

**}**

# Arduino and Raspberry Pi Working Together

# With I2C

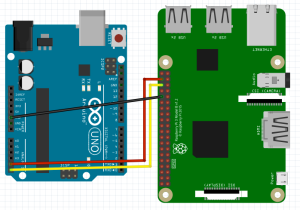
The easiest way to [connect our Arduino board to our Raspberry Pi](https://dzone.com/articles/arduino-and-raspberry-pi-working-together) is using a USB cable, but sometimes this communication is a nightmare, especially because there isn’t any clock signal to synchronize our devices and we must rely on the bitrate. There are different ways to connect our Arduino and our Raspberry Pi, such as I2C, SPI, and serial over GPIO. Today, we’re going to speak about I2C, especially because it’s pretty straightforward if we take care of a couple of things. Let’s start.

**I2C** uses two lines — **SDA** (data) and **SCL** (clock) — in addition to **GND** (ground). SDA is bidirectional, so we need to ensure one way or another who is sending data (master or slave). With I2C, only the master can start communications. The master also controls the clock signal. Each device has a 7-bit direction, so we can connect 128 devices to the same bus.

If we want to connect an Arduino board and a Raspberry Pi we must ensure that the Raspberry Pi is the master. That’s because the Arduino works with 5V while the Raspberry Pi works with 3.3V. That means that we need to use pull-up resistors if we don’t want to destroy our Raspberry Pi. But the Raspberry Pi has 1k8 ohms resistors to the 3.3-volt power rail, so we can connect both devices (if we connect other I2C devices to the bus, they must have their pull-up resistors removed)

That's all we need to connect our Raspberry Pi to our Arduino board.

* RPi SDA to Arduino analog 4
* RPi SCL to Arduino analog 5
* RPi GND to Arduino GND

[](https://gonzalo123.files.wordpress.com/2017/04/untitled_sketch_fzz_-_fritzing_-__breadboard_view_.png)

Now we are going to build a simple prototype. The Raspberry Pi will blink one LED (GPIO17) each second while also sending a message (via I2C) to the Arduino to blink another LED. That’s the Python part:

import RPi.GPIO as gpio

import smbus

import time

import sys

bus = smbus.SMBus(1)

address = 0x04

def main():

gpio.setmode(gpio.BCM)

gpio.setup(17, gpio.OUT)

status = False

while 1:

gpio.output(17, status)

status = not status

bus.write\_byte(address, 1 if status else 0)

print "Arduino answer to RPI:", bus.read\_byte(address)

time.sleep(1)

if \_\_name\_\_ == '\_\_main\_\_':

try:

main()

except KeyboardInterrupt:

print 'Interrupted'

gpio.cleanup()

sys.exit(0)

And finally, the Arduino program. Arduino also answers to the Raspberry Pi with the value that it has been sent, and the Raspberry Pi will log the answer within the console.

#include <Wire.h>

#define SLAVE\_ADDRESS 0x04

#define LED 13

int number = 0;

void setup() {

pinMode(LED, OUTPUT);

Serial.begin(9600);

Wire.begin(SLAVE\_ADDRESS);

Wire.onReceive(receiveData);

Wire.onRequest(sendData);

Serial.println("Ready!");

}

void loop() {

delay(100);

}

void receiveData(int byteCount) {

Serial.print("receiveData");

while (Wire.available()) {

number = Wire.read();

Serial.print("data received: ");

Serial.println(number);

if (number == 1) {

Serial.println(" LED ON");

digitalWrite(LED, HIGH);

} else {

Serial.println(" LED OFF");

digitalWrite(LED, LOW);

}

}

}

void sendData() {

Wire.write(number);

}

Hardware:

* Arduino UNO
* Raspberry Pi
* Two LEDs and two resistors

Arduino Program :

|  |
| --- |
| #include <Wire.h> |
|  |  |
|  | #define SLAVE\_ADDRESS 0x04 |
|  | #define LED 13 |
|  |  |
|  | int number = 0; |
|  |  |
|  | void setup() { |
|  | pinMode(LED, OUTPUT); |
|  | Serial.begin(9600); |
|  | Wire.begin(SLAVE\_ADDRESS); |
|  | Wire.onReceive(receiveData); |
|  | Wire.onRequest(sendData); |
|  |  |
|  | Serial.println("Ready!"); |
|  | } |
|  |  |
|  | void loop() { |
|  | delay(100); |
|  | } |
|  |  |
|  | void receiveData(int byteCount) { |
|  | Serial.print("receiveData"); |
|  | while (Wire.available()) { |
|  | number = Wire.read(); |
|  | Serial.print("data received: "); |
|  | Serial.println(number); |
|  |  |
|  | if (number == 1) { |
|  | Serial.println(" LED ON"); |
|  | digitalWrite(LED, HIGH); |
|  | } else { |
|  | Serial.println(" LED OFF"); |
|  | digitalWrite(LED, LOW); |
|  | } |
|  | } |
|  | } |
|  |  |
|  | void sendData() { |
|  | Wire.write(number); |
|  | } |

Rasp program :

import RPi.GPIO as gpio

import smbus

import time

import sys

bus = smbus.SMBus(1)

address = 0x04

def main():

gpio.setmode(gpio.BCM)

gpio.setup(17, gpio.OUT)

status = False

while 1:

gpio.output(17, status)

status = not status

bus.write\_byte(address, 1 if status else 0)

print "Arduino answer to RPI:", bus.read\_byte(address)

time.sleep(1)

if \_\_name\_\_ == '\_\_main\_\_':

try:

main()

except KeyboardInterrupt:

print 'Interrupted'

gpio.cleanup()

sys.exit(0)