Tutorial: How To Control the Tower Pro SG90 Servo with Arduino UNO

笔记本：	Arduino课程		
创建时间：	2017/9/13 9:21	**更新时间：**	2017/9/15 22:40
作者：	王景夏		
标签：	第一期		
URL：	https://www.intorobotics.com/tutorial-how-to-control-the-tower-pro-sg90-servo-with-arduino-uno/		

Tutorial: How To Control the Tower Pro SG90 Servo with Arduino UNO

来源网址： https://www.intorobotics.com/tutorial-how-to-control-the-tower-pro-sg90-servo-with-arduino-uno/
作者： 王景夏

January 10, 2016

Tutorial: How To Control the Tower Pro SG90 Servo with Arduino UNO

I write this tutorial to show you how to control the direction, position, and speed of the SG90 9G Micro servo motor with the Arduino UNO board. I know that if you're a hobbyist with some experience in robots is very easy to control this servo motor, but like always, you couldn't miss anything in the field. Even so, if this tutorial is boring for you, here is a list of cheap projects for Arduino UNO.

Tutorial: How To Control the Tower Pro SG90 Servo with Arduino UNO

How the tutorial is organized:

1. I start the tutorial with a short overview of the parts needed to control the servo motor;
2. Then I did an overview of the Tower Pro SG90 servo;
3. I break down into components the servo motor to show you the parts;
4. Finally, I went to the practice part to show you how to wire TowerPro SG90, writes the Arduino sketch, and some hacks for the servo motor.

A servo motor allows a precise control of the angular position, velocity, and acceleration. It's like you're at the steering wheel of your car. You control precisely the speed of the car and the direction.

This very strict control of the angular position, velocity, and acceleration can't be done without a sensor for position feedback.

This sensor sounds the alarm when the motor is spinning. But even so, there is something more sophisticated that controls all the stages of the servo motor. It’s a dedicated controller that makes the tiny things inside the servo to move with military precision.

A feedback sensor, a controller, and a motor. I’m just asking – why we need a servo motor? These servos are essential parts if we need to control the position of objects, rotate sensors, move arms and legs, drive wheels and tracks, and more.

1. Hardware Required

This is not a complex project, but we still need some parts. Luckily, all the components are cheap and available worldwide.

To control the TowerPro SG90 servo, you will need the following parts:

- 1 x TowerPro SG90 servo motor (price ~$5)
- 1 x Arduino UNO (price ~$21)
- 1 x Breadboard (price ~$4.4/piece)
- 6 x jumper wires (price ~$6.7)

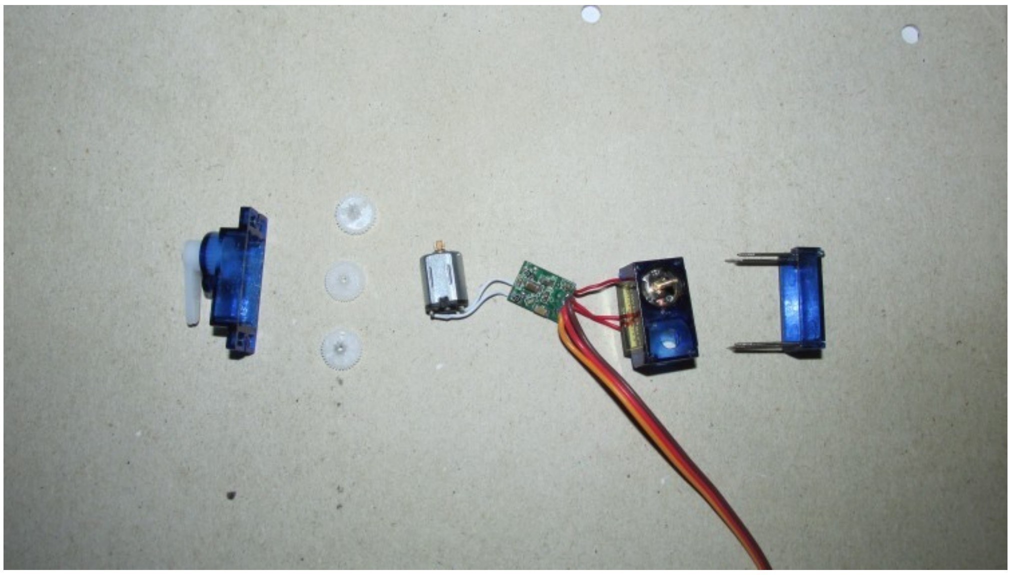
2. The Tower Pro SG90 Servo

The Tower Pro SG90 servo is one of the cheapest servo motors that you can find on the market. Even it’s cheap, less than $5, don’t try to rotate the servo motor by hand because this may damage the motor.

Let’s make a short overview of the SG90 specifications.

You need torque to control the position of an object, for example, and this little box that weight 0.32 oz (9.0 g) can provide at 4.8V a torque of 25.0 oz-in (1.80 kg-cm). At 4.8V, the speed of the servo is 0.12 sec/60°. All these specifications are really impressive for this little plastic box. Well, there are other high-end servos with a bit more muscle than the SG90 servo, but these high-end servos can’t beat the price of the SG90 servo. SG90 is cheap enough to throw away when they break.

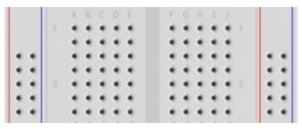
3. Take a look inside of the SG90 micro servo

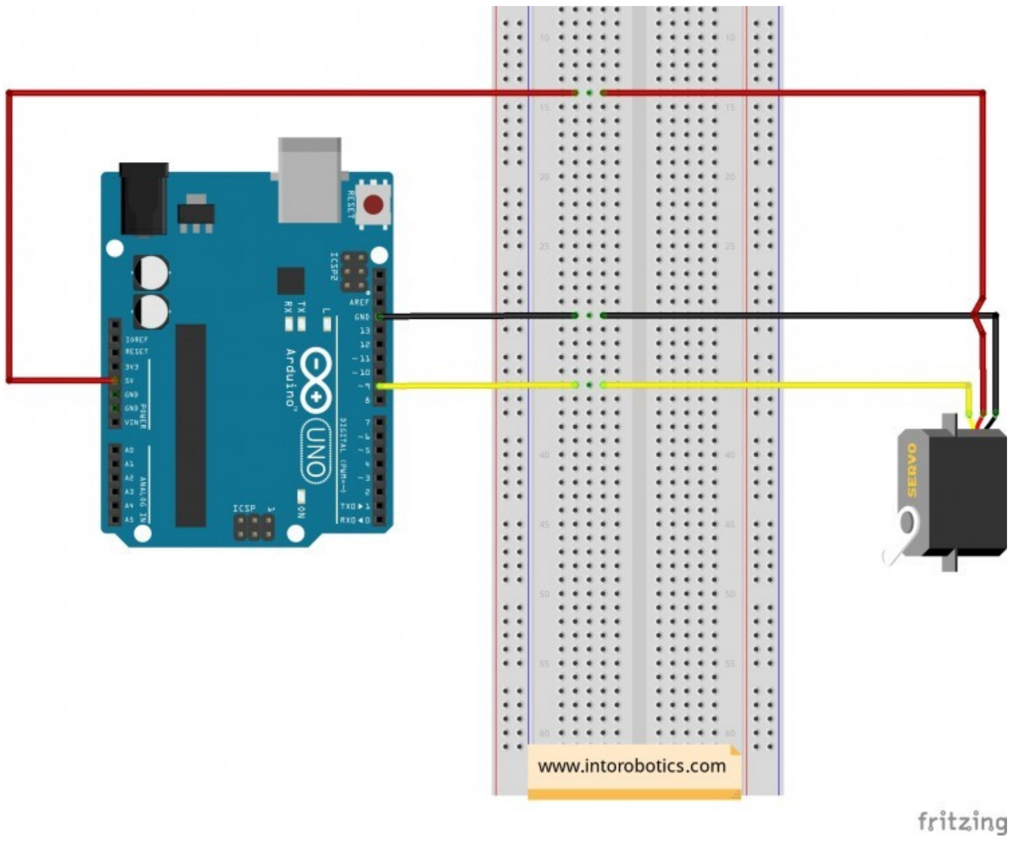

Inside SG90 servo

Inside the micro servo, you will find the pieces from the above image. The top cover hosts the plastic gears while the middle cover hosts a DC motor, a controller, and the potentiometer.

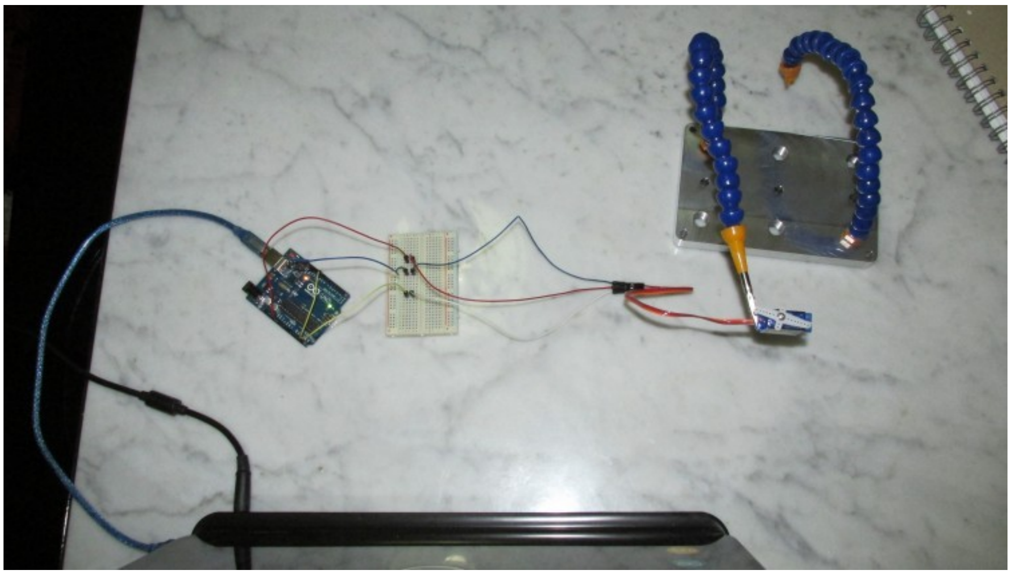
4. How to wire TowerPro SG90, the Arduino sketch, and some hacks

Considering that you already have an Arduino UNO, the SG90 servo, and the six wires, your circuit should look like this (plus bonus the scheme):





Arduino SG Servo Motor Control Schema



Arduino SG Servo Motor Control Circuit

It would be something to note here. The servo motor has three leads, with one more than a DC motor. Each lead has a color code. So you have to connect the brown wire from the micro servo to the GND pin on the Arduino. Connect the red wire from the servo to the +5V on the Arduino. And finally, connect the orange wire from the SG90 servo to a digital pin (pin 9) on the Arduino.

Arduino makes the things simple. In this tutorial, I use the SG90 servo powered directly from the Arduino via USB. And you can do the same.

The Arduino sketch

Below you can find the Arduino sketch that controls the servo's direction, the position of the motor and the speed of the SG90 servo. Before reaching the Arduino code, I want to write few words about the file. This library makes our life easier. It contains all the functions required for controlling the SG90 servo.

the functions required for controlling the SG90 servo.

```
/*
Into Robotics
*/

#include servo.h  //add '<' and '>' before and after servo.h

int servoPin = 9;

Servo servo;

int servoAngle = 0;   // servo position in degrees

void setup()
{
  Serial.begin(9600);
  servo.attach(servoPin);
}


void loop()
{
//control the servo's direction and the position of the motor

   servo.write(45);      // Turn SG90 servo Left to 45 degrees
   delay(1000);          // Wait 1 second
   servo.write(90);      // Turn SG90 servo back to 90 degrees (center position)
   delay(1000);          // Wait 1 second
   servo.write(135);     // Turn SG90 servo Right to 135 degrees
   delay(1000);          // Wait 1 second
   servo.write(90);      // Turn SG90 servo back to 90 degrees (center position)
   delay(1000);

//end control the servo's direction and the position of the motor


//control the servo's speed

//if you change the delay value (from example change 50 to 10), the speed of the servo
changes
  for(servoAngle = 0; servoAngle < 180; servoAngle++)  //move the micro servo from 0
degrees to 180 degrees
  {
    servo.write(servoAngle);
    delay(50);
  }

  for(servoAngle = 180; servoAngle > 0; servoAngle--)  //now move back the micro servo
from 0 degrees to 180 degrees
  {
    servo.write(servoAngle);
    delay(10);
  }
  //end control the servo's speed
}
```

You can also hack the SG 90 micro servo

The SG90 micro servo can turn your robot wheel. And this is great since the shapes of the servo box help you attach it to a robot chassis. But there is a problem with the rotation of the SG90 servo. It's about the rotation that reach a maximum 180 degrees.

In this case, you need to hack it to have a continuous rotation. This is a delicate operation that requires time and cautions. Before proceeding any operation, make sure you have extra money or other micro servo if you're doing this wrong.

This tutorial shows you step-by-step how to hack a Tower Pro SG 90 micro servo for continuous rotation.

And in the end, I hope this tutorial helps you guys learn how to control the Tower Pro SG 90 micro servo with Arduino UNO. Also, don't forget to share the tutorial on social networks to help others learn to control their micro servo.

Disclaimer: This post contains affiliate links.

Thanks for reading. And before you go… If you found this article helpful, share the article on Facebook and Twitter so other people can benefit from it too.

Tutorials

Tagged: Arduino, How To, Tutorials

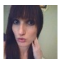
Commerce Content is independent of articles and advertising, and if you buy something through our posts, I may get a small share of the sale.

Create a Bluetooth HC-06 Module With Arduino - DZone IoT

来源网址： https://dzone.com/articles/bluetooth-hc-06-module-with-arduino
作者： 王景夏

Create a Bluetooth HC-06 Module With Arduino

Learn how to wire and program a module to connect to Bluetooth so you can send and receive data.


by Maddie Abboud · Oct. 04, 16 · IoT Zone

鹗 Comment (1) 耀 2,699 Views

赌 Like (2) Save Tweet

Join the DZone community and get the full member experience. JOIN FOR FREE

Address your IoT software testing needs – improve quality, security, safety, and compliance across the development lifecycle.

A little bit ago I grabbed a cheap HC-06 Bluetooth transceiver for $6 on Amazon for my electronics project. It was fairly simple to set up but I did run into a few hitches and a (lack of information to fix them), so I'm going to detail some of my experience for you so you can hopefully avoid the same pitfalls.

Here's a cheap one for $6.50 on Amazon. Note that there are many sellers on Amazon and Ebay selling HC-06s as HC-05s so if you can't get AT commands to work or only basic ones work then you probably don't have a HC-05 module.

Step #1: Wiring it Up

The first step, of course, is to wire up the Bluetooth leads to your Arduino pins. RX goes to your Arduino TX and TX goes to your Arduino's RX — remember, they're opposite because the Bluetooth chip is sending on TX, so the Arduino receives that on a RX pin.

Arduino Uno pins	
RX	**TX**
0	1

Arduino Mega pins	
RX	**TX**
19	18
17	16
15	14

VCC/3.3V goes to 3.3V — **not 5V**: Using 5V is likely to damage your Bluetooth chip, but it could probably stand it for a brief moment if you do accidentally connect it to 5V.

Once it's connected and you turn on your Arduino, an LED on the Bluetooth board should start blinking.

Step #2: Setup Code and AT Command Configuration

Next, we need to write some code so we can use this thing. Bluetooth devices can be configured with these various AT commands and SoftwareSerial, a standard Arduino library takes care of communication for us quite neatly.

Note the cheaper Bluetooth boards like the one I'm using are usually HC-06, not HC-05. HC-05 has more AT commands — and also some different ones — that will not work for yours. HC-06 will also not work if you include new line characters. There's a bunch of example code out there that includes newline characters and also unsupported AT commands that will not work at all for your HC-06 firmware.

So below is some simple setup code that initializes the Bluetooth device and tells it to change its broadcast name to My-Sweet project. Note the delays in sending the AT config commands. In a HC-05 user guide I ran into, the delays are stated as necessary, and it appeared to be flakier without them.

```
#include <SPI.h>
#include <EEPROM.h>
```

```
#include <TouchScreen.h>
#define BLUETOOTH_RX 10
#define BLUETOOTH_TX 11
SoftwareSerial BT(BLUETOOTH_RX, BLUETOOTH_TX);
void setup(void) {
    // int passed here should match your Bluetooth's set baud rate.
    // Default is almost always 9600
    BT.begin(9600);
    delay(500);
    BT.print("AT");
    delay(500);
    BT.print("AT+VERSION");
    delay(500);
    // renames your BT device name to My-Sweet-Project
    String nameCommand = "AT+NAME" + "My-Sweet-Project";
    BT.print(nameCommand);
    delay(500);
    if (BT.available() > 0)
        Serial.println(BT.readString());
}
```

Here’s a table of HC-06 AT commands (which is quite limited compared to HC-05)

AT	Responds with the connection status, also seemed like you had to run it first to get other AT commands to work
AT+VERSION	Responds with device version information
AT+NAME	Changes the broadcast name of the device, name can be up to 31 chars. ex.: AT+NAMEYOUR-NAME-HERE
AT+PIN	Changes pairing password for the device, 4 bits max. ex.: AT+PIN0420
AT+BAUD	Sets the baud rate input is 1-8 which corresponds to baud rates seen in the next table. ex.: AT+BAUD4

AT+BAUD flags and corresponding baud rate	
1	1200
2	2400
3	4800
4	9600
5	19200
6	38400
7	57600
8	115200

Remember, HC-06 AT commands have no line endings at the end and no spaces between the command name and the input. It's AT+NAMEYOUR_NAME, not AT+NAME YOUR_NAME. If you see a command like AT+ORGL or AT+UART, the code you have is for HC-05.

Also note that the higher baud rates are unlikely to work well enough for these cheap devices and are not necessary for many types of projects. Furthermore, the cheap ones have a tendency of getting stuck on the baud rate if you try to change it and won't let you change the baud rate back to the original. I got mine stuck this way when testing out AT+BAUD, and I also initially forgot to change the software serial baud rate to the new one. **Remember to change it if you do use AT+BAUD**.

bluetooth-board-2

Step #3: Sending Data Back and Forth

Next, we should test out sending and receiving data. I suggest grabbing BlueTerm from the Android Play Store. It's a good ad-free app for sending and receiving raw Bluetooth data and is good for debugging. After you connect your device to a phone or something else, the LED should stop blinking and just remain steady. **Note**: It will only do this if connected in an app, not if it's just simply paired.

So to send data, you just call print on your SoftwareSerial object — perhaps in your main loop:

```
void loop(void) {
    BT.print("Hello Bluetooth!");
    BT.flush();
    delay(2000);
}
```

;

If you have your device paired with BlueTerm correctly, you should start seeing, “Hello Bluetooth!” repeated every 2 seconds.

To receive, use code like that below and type in a message in BlueTerm or other Bluetooth app:

```
void loop(void) {
    BT.print("Hello Bluetooth!");
    BT.flush();
    delay(2000);
    if (BT.available() > 0) {
        Serial.println("Message recieved!");
        Serial.println(BT.readString());
    }
}
```

Other Notes

For a more complete implementation, you can take a look at my Bluetooth project on GitHub here. The project includes methods for sending data in JSON format with the Bluetooth transceiver in the BluetoothUIController.cpp file and also includes a companion Android app that receives those JSON messages and can send commands back down.

Thank you for reading!

Arduino - ShiftOut

来源网址： https://www.arduino.cc/en/Tutorial/ShiftOut
作者： 王景夏

TUTORIALS > Foundation > Serial to Parallel conversion

Serial to Parallel Shifting-Out with a 74HC595

Shifting Out & the 595 chip

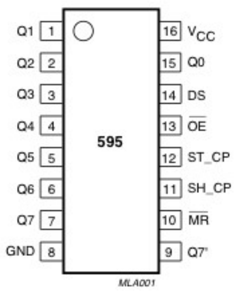
At sometime or another you may run out of pins on your Arduino board and need to extend it with shift registers. This example is based on the 74HC595. The datasheet refers to the 74HC595 as an "8-bit serial-in, serial or parallel-out shift register with output latches; 3-state." In other words, you can use it to control 8 outputs at a time while only taking up a few pins on your microcontroller. You can link multiple registers together to extend your output even more. (Users may also wish to search for other driver chips with "595" or "596" in their part numbers, there are many. The STP16C596 for example will drive 16 LED's and eliminates the series resistors with built-in constant current sources.)

How this all works is through something called "synchronous serial communication," i.e. you can pulse one pin up and down thereby communicating a data byte to the register bit by bit. It's by pulsing second pin, the clock pin, that you delineate between bits. This is in contrast to using the "asynchronous serial communication" of the Serial.begin() function which relies on the sender and the receiver to be set independently to an agreed upon specified data rate. Once the whole byte is transmitted to the register the HIGH or LOW messages held in each bit get parceled out to each of the individual output pins. This is the "parallel output" part, having all the pins do what you want them to do all at once.

The "serial output" part of this component comes from its extra pin which can pass the serial information received from the microcontroller out again unchanged. This means you can transmit 16 bits in a row (2 bytes) and the first 8 will flow through the first register into the second register and be expressed there. You can learn to do that from the second example.

"3 states" refers to the fact that you can set the output pins as either high, low or "high impedance." Unlike the HIGH and LOW states, you can"t set pins to their high impedance state individually. You can only set the whole chip together. This is a pretty specialized thing to do -- Think of an LED array that might need to be controlled by completely different microcontrollers depending on a specific mode setting built into your project. Neither example takes advantage of this feature and you won"t usually need to worry about getting a chip that has it.

Here is a table explaining the pin-outs adapted from the Phillip's datasheet.

 Q1 1 Q2 2 Q3 3 Q4 4 Q5 5 Q6 6 Q7 7 GND 8 595 16 VCC 15 Q0 14 DS 13 OE 12 ST_CP 11 SH_CP 10 MR 9 Q7' MLA001	PINS 1-7, 15	Q0 " Q7	Output Pins
	PIN 8	GND	Ground, Vss
	PIN 9	Q7"	Serial Out
	PIN 10	MR	Master Reclear, active low
	PIN 11	SH_CP	Shift register clock pin
	PIN 12	ST_CP	Storage register clock pin (latch pin)
	PIN 13	OE	Output enable, active low
	PIN 14	DS	Serial data input
	PIN 16	Vcc	Positive supply voltage

Example 1: One Shift Register

The first step is to extend your Arduino with one shift register.

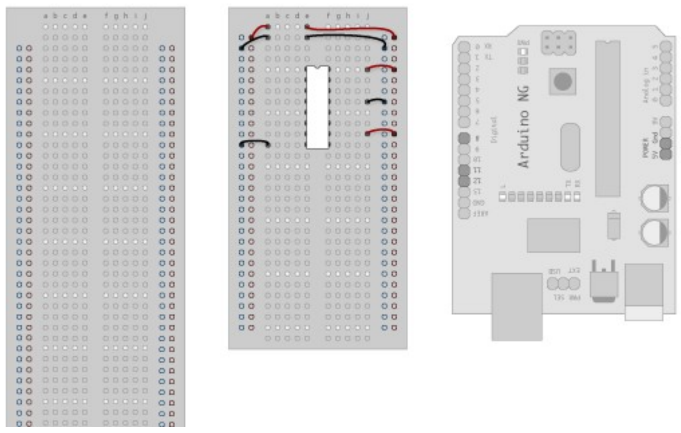
The Circuit

1. Turning it on

Make the following connections:

- GND (pin 8) to ground,
- Vcc (pin 16) to 5V
- OE (pin 13) to ground
- MR (pin 10) to 5V

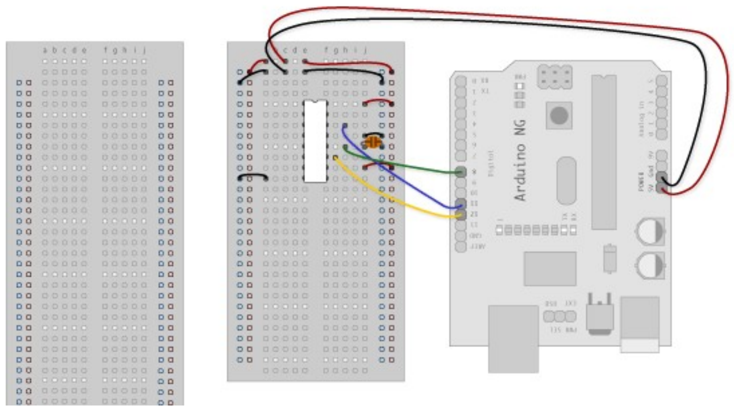
This set up makes all of the output pins active and addressable all the time. The one flaw of this set up is that you end up with the lights turning on to their last state or something arbitrary every time you first power up the circuit before the program starts to run. You can get around this by controlling the MR and OE pins from your Arduino board too, but this way will work and leave you with more open pins.



2. Connect to Arduino

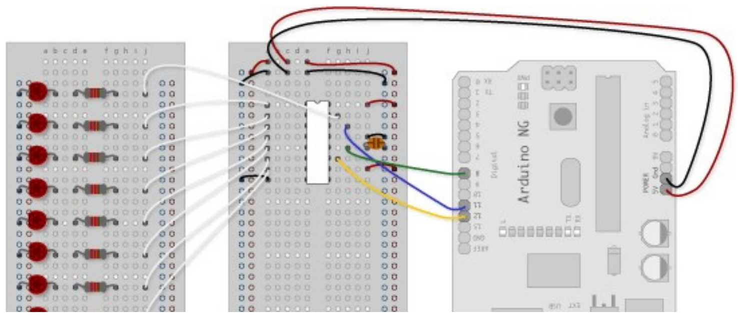
- DS (pin 14) to Ardunio DigitalPin 11 (blue wire)
- SH_CP (pin 11) to to Ardunio DigitalPin 12 (yellow wire)
- ST_CP (pin 12) to Ardunio DigitalPin 8 (green wire)

From now on those will be refered to as the dataPin, the clockPin and the latchPin respectively. Notice the 0.1"f capacitor on the latchPin, if you have some flicker when the latch pin pulses you can use a capacitor to even it out.

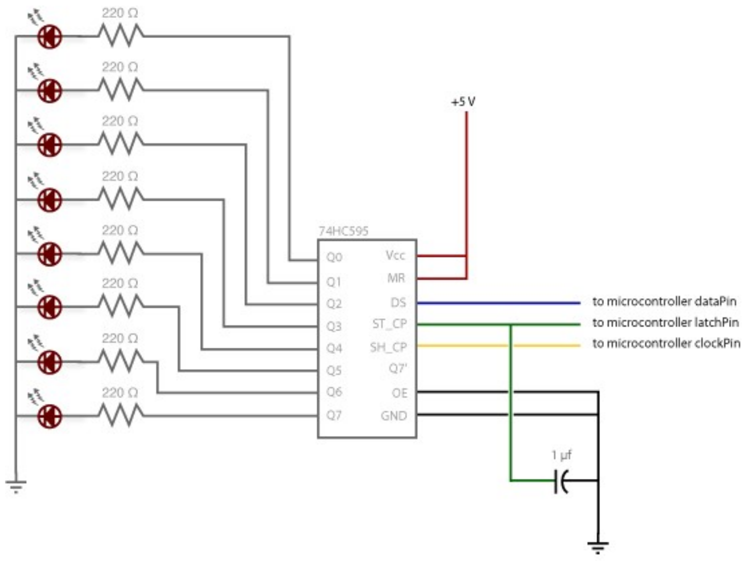


3. Add 8 LEDs.

In this case you should connect the cathode (short pin) of each LED to a common ground, and the anode (long pin) of each LED to its respective shift register output pin. Using the shift register to supply power like this is called *sourcing current.* Some shift registers can't source current, they can only do what is called *sinking current.* If you have one of those it means you will have to flip the direction of the LEDs, putting the anodes directly to power and the cathodes (ground pins) to the shift register outputs. You should check the your specific datasheet if you aren"t using a 595 series chip. Don"t forget to add a 220-ohm resistor in series to protect the LEDs from being overloaded.



Circuit Diagram

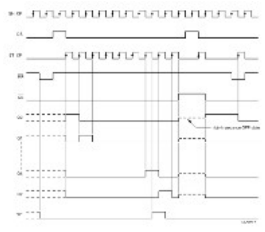

220 Ω
+5 V
74HC595
Q0
Q1
Q2
Q3
Q4
Q5
Q6
Q7
Vcc
MR
DS
ST_CP
SH_CP
Q7'
OE
GND
to microcontroller dataPin
to microcontroller latchPin
to microcontroller clockPin
1 µf

The Code

Here are three code examples. The first is just some "hello world" code that simply outputs a byte value from 0 to 255. The second program lights one LED at a time. The third cycles through an array.

The code is based on two pieces of information in the datasheet: the timing diagram and the logic table. The logic table is what tells you that basically everything important happens on an up beat. When the clockPin goes from low to high, the shift register reads the state of the data pin. As the data gets shifted in it is saved in an internal memory register. When the latchPin goes from low to high the sent data gets moved from the shift registers aforementioned memory register into the output pins, lighting the LEDs.

Code Sample 1.1 Hello World
Code Sample 1.2 One by One
Code Sample 1.3 Using an array


595 Timing Diagram

595 Logic Table

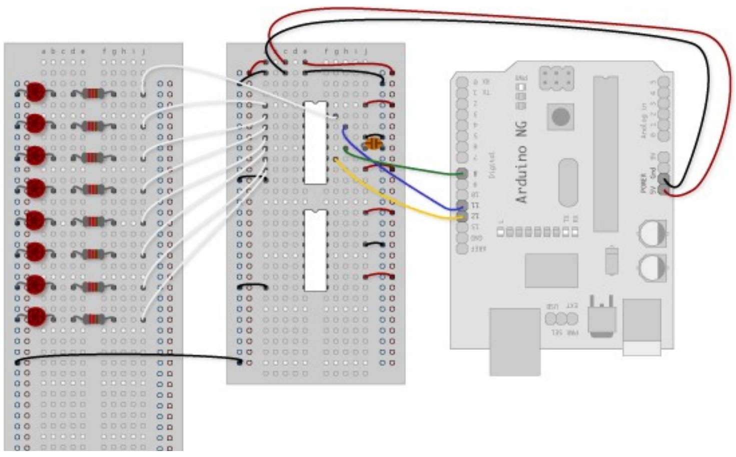
Example 2

In this example you'll add a second shift register, doubling the number of output pins you have while still using the same number of pins from the Arduino.

The Circuit

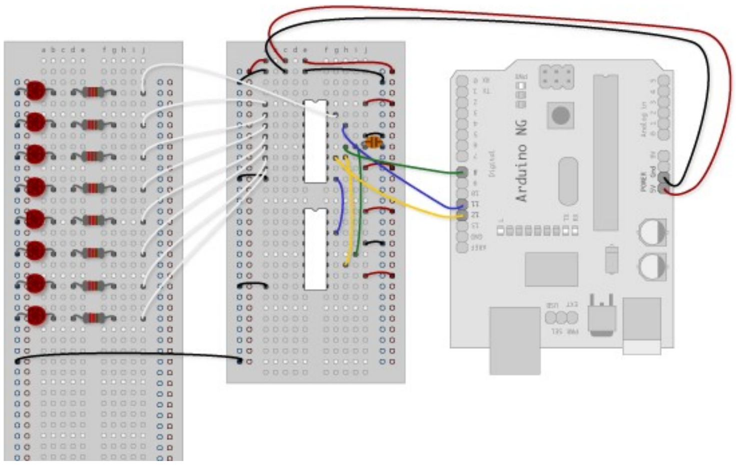
1. Add a second shift register.

Starting from the previous example, you should put a second shift register on the board. It should have the same leads to power and ground.



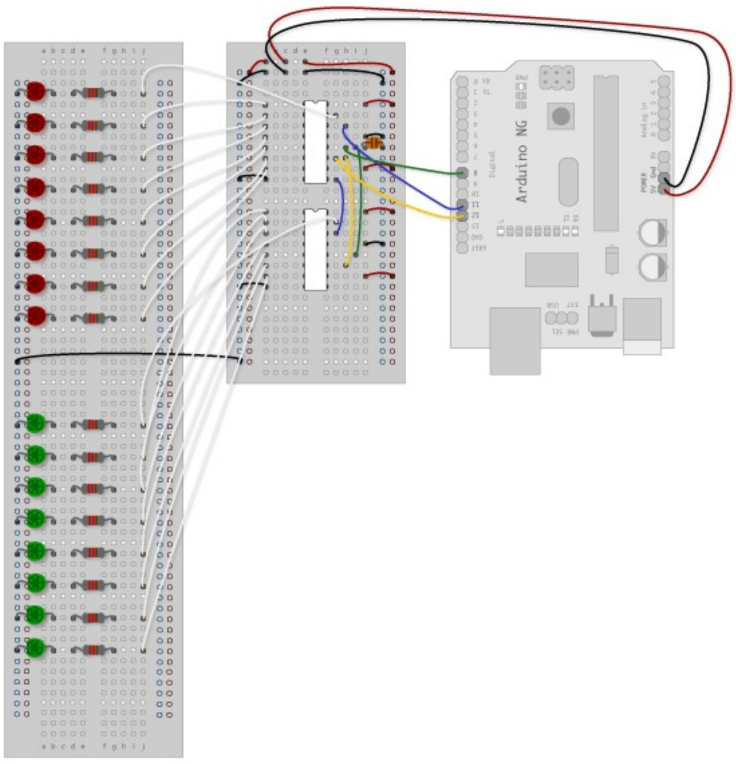
2. Connect the 2 registers.

Two of these connections simply extend the same clock and latch signal from the Arduino to the second shift register (yellow and green wires). The blue wire is going from the serial out pin (pin 9) of the first shift register to the serial data input (pin 14) of the second register.

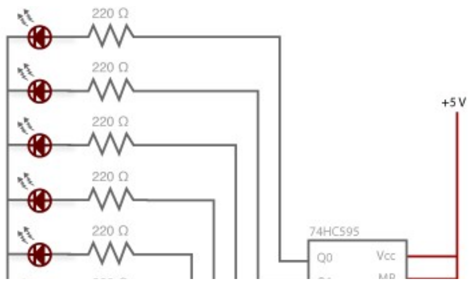


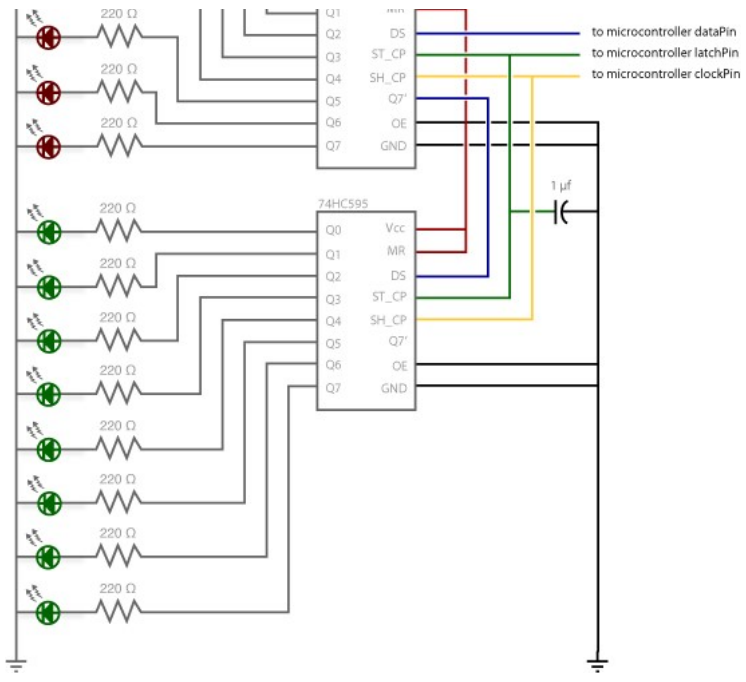
3. Add a second set of LEDs.

In this case I added green ones so when reading the code it is clear which byte is going to which set of LEDs



Circuit Diagram





220 Ω
Q1
Q2
Q3
Q4
Q5
Q6
Q7
DS
ST_CP
SH_CP
Q7'
OE
GND
to microcontroller dataPin
to microcontroller latchPin
to microcontroller clockPin
1 µf
74HC595
Q0
Vcc
MR

The Code

Here again are three code samples. If you are curious, you might want to try the samples from the first example with this circuit set up just to see what happens.

Code Sample 2.1 Dual Binary Counters
There is only one extra line of code compared to the first code sample from Example 1. It sends out a second byte. This forces the first shift register, the one directly attached to the Arduino, to pass the first byte sent through to the second register, lighting the green LEDs. The second byte will then show up on the red LEDs.

Code Sample 2.2 2 Byte One By One
Comparing this code to the similar code from Example 1 you see that a little bit more has had to change. The blinkAll() function has been changed to the blinkAll_2Bytes() function to reflect the fact that now there are 16 LEDs to control. Also, in version 1 the pulsings of the latchPin were situated inside the subfunctions lightShiftPinA and lightShiftPinB(). Here they need to be moved back into the main loop to accommodate needing to run each subfunction twice in a row, once for the green LEDs and once for the red ones.

Code Sample 2.3 - Dual Defined Arrays
Like sample 2.2, sample 2.3 also takes advantage of the new blinkAll_2bytes() function. 2.3's big difference from sample 1.3 is only that instead of just a single variable called "data" and a single array called "dataArray" you have to have a dataRED, a dataGREEN, dataArrayRED, dataArrayGREEN defined up front. This means that line

```
  data = dataArray[j];
```

becomes

```
  dataRED = dataArrayRED[j];
dataGREEN = dataArrayGREEN[j];
```

and

```
  shiftOut(dataPin, clockPin, data);
```

becomes

```
  shiftOut(dataPin, clockPin, dataGREEN);
shiftOut(dataPin, clockPin, dataRED);
```

Started by Carlyn Maw and Tom Igoe Nov, 06

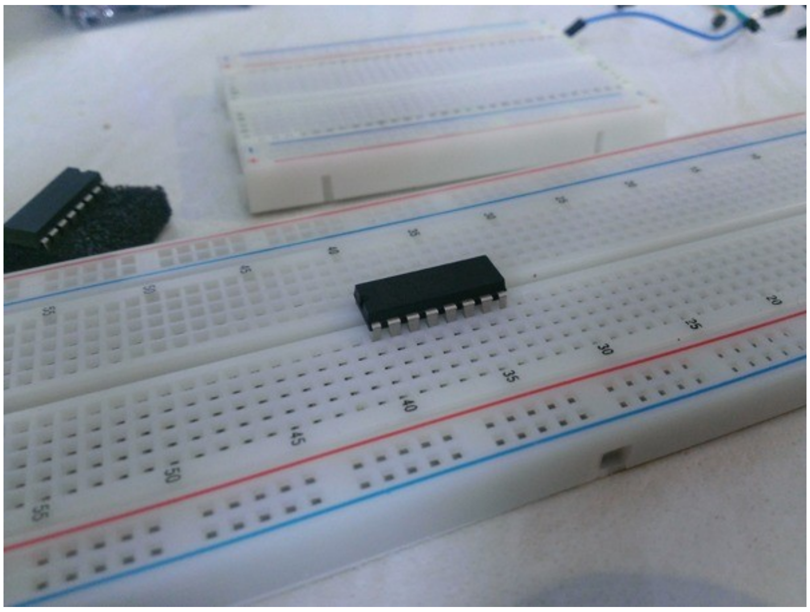
Using a 74HC595 Shift Register with an Arduino Uno

来源网址： http://www.rastating.com/using-a-74hc595-shift-register-with-an-arduino-uno/
作者： 王景夏

USING A 74HC595 SHIFT REGISTER WITH AN ARDUINO UNO

PUBLISHED MAY 27TH 2013 BY ROB CARR

At one point or another, you're inevitability going to run into the problem of not having enough pins on your Arduino to meet the requirements of your project or prototype. The solution to this problem? A shift register!



A shift register allows you to expand the number of pins you can use from your Arduino (or any micro controller for that matter) by using what is known as bit-shifting. If you have much experience with programming, there's a good chance you have come across bit-shifting previously. For the purpose of this guide though, we'll assume no prior knowledge.

In order to complete this guide you will need the following parts:

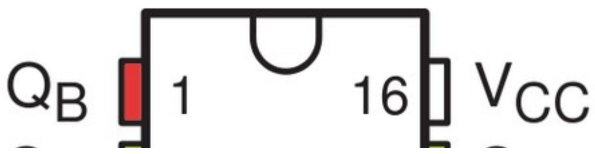
- An Arduino Uno
- A breadboard – I'd recommend two, as there will be a lot of cables
- A 74HC595 shift register
- 8 LEDs
- 8 resistors – 220 ohm should suffice
- A lot of jumper cables

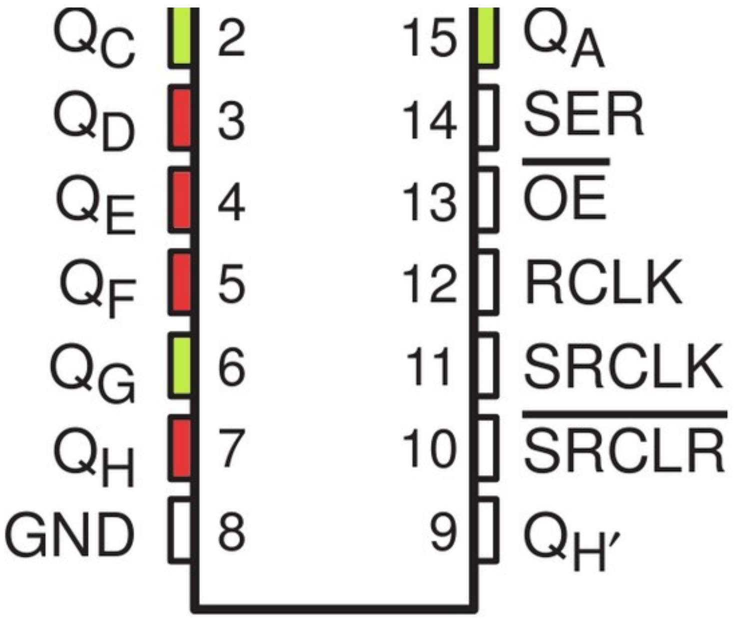
In this guide I'll be using the 74HC595 8-bit shift register, which you can pick up from most places at a very reasonable price. This shift register will provide us with a total of eight extra pins to use (five if you subtract the three pins we need to connect to the Arduino from the shift register itself).

Before we begin wiring up the chip, let's take a moment to go over how this process works.

The first thing that should be cleared up is what "bits" are, for those of you who aren't familiar with binary. When we refer to a "bit", we are referring to one of the numbers that make up the binary value. Unlike normal numbers though, we typically consider the first bit to be the right most one. So, if we take the binary value **10100010**, the first bit is actually **0**, and the eighth bit is **1**. It should also be noted, in case it wasn't implied, each bit can only be 0 or 1.

The chip contains eight pins that we can use for output, each of which is associated with a bit in the register. In the case of the 74HC595 IC, we refer to these as QA through to QH. In order to write to these outputs via the Arduino, we have to send a binary value to the shift register, and from that number the shift register can figure out which outputs to use. For example, if we sent the binary value **10100010**, the pins highlighted in green in the image below would be active and the ones highlighted in red would be inactive.

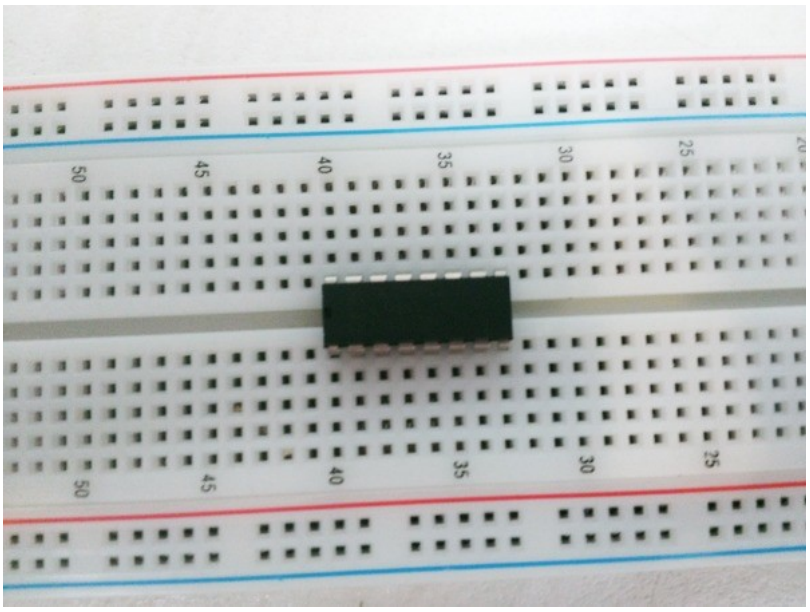




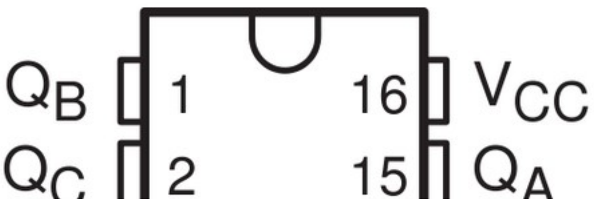
This means that the right most bit that w e specify maps to QH, and the left most bit maps to QA. An output is considered active w hen the bit mapped to it is set to 1. It is important to remember this, as otherw ise you w ill have a very hard time know ing w hich pins you are using!

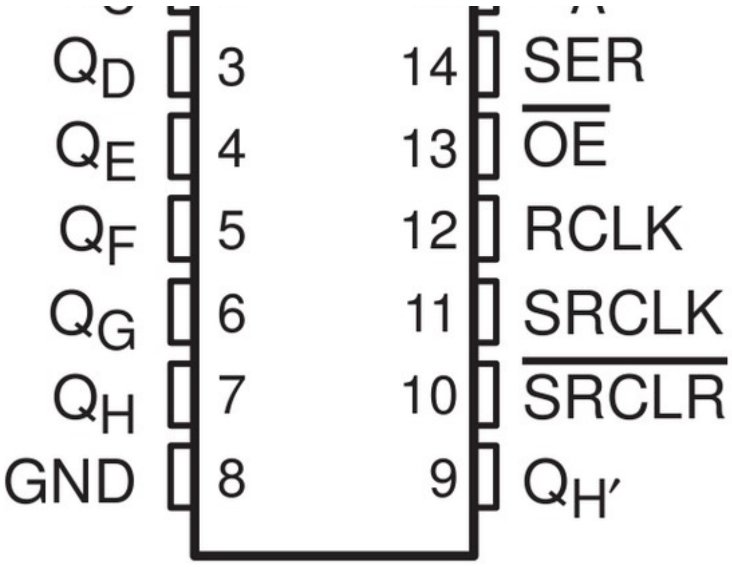
Now that w e have a basic understanding of how w e use bit shifting to specify w hich pins to use, w e can begin hooking it up to our Arduino!

Start by placing the shift register on to your breadboard, ensuring each side of the IC is on a separate side of the breadboard, as per below .



With the notch facing upw ards, the pins are 1-8 dow n the left hand side from top to bottom and 16 – 9 dow n the right hand side from top to bottom as can be seen in the illustration below .





To start w ith let's connect pins 16 (VCC) and 10 (SRCLR) to the 5v pin on the Arduino and connect pins 8 (GND) and 13 (OE) to the Gnd pin on the Arduino. Pin 13 (OE) is used to enable the outputs, as this is an active low pin w e can just connect this directly to ground.

Next w e need to connect the three pins that w e w ill control the shift register w ith:

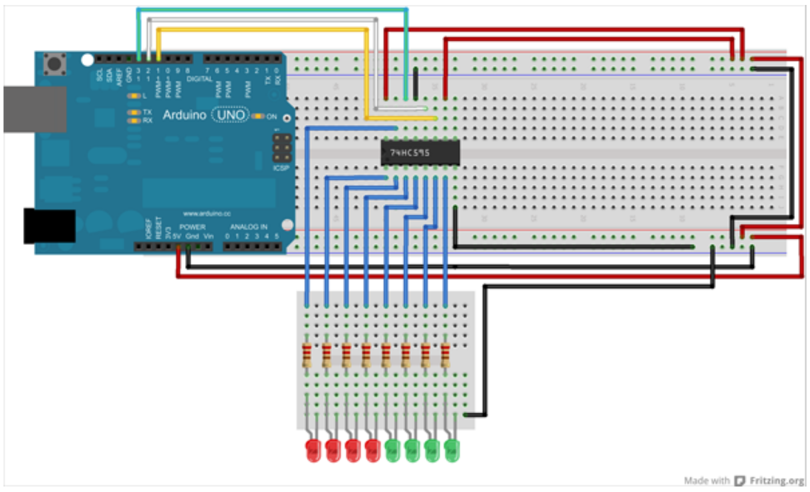
- Pin 11 (SRCLK) of the shift register to pin 11 on the Arduino – this w ill be referred to as the "clock pin"
- Pin 12 (RCLK) of the shift register to pin 12 on the Arduino – this w ill be referred to as the "latch pin"
- Pin 14 (SER) of the shift register to pin 13 on the Arduino – this w ill be referred to as the "data pin"

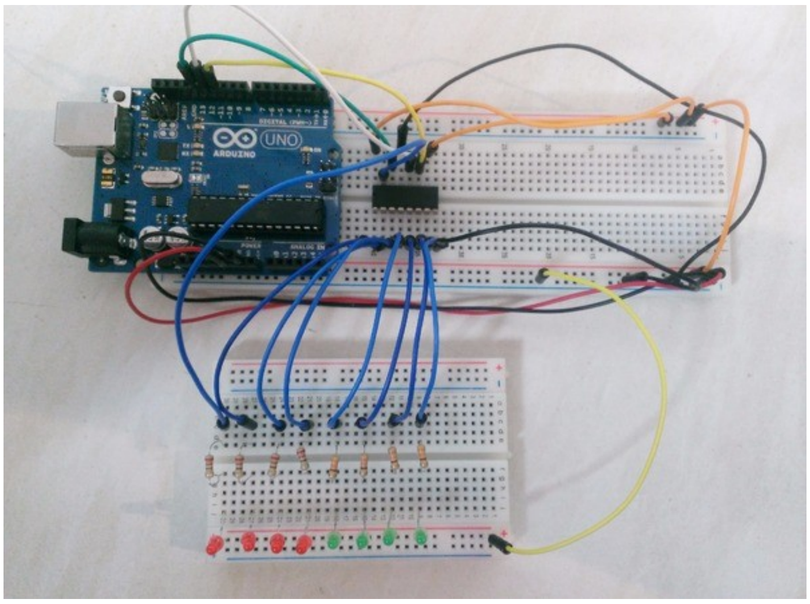
All three of these pins are used in order to do the bit shifting that w as mentioned earlier in this guide. Thankfully Arduino provide a helper function specifically for shift registers called **shiftOut**, w hich w ill handle pretty much everything for us; but w e'll get back to that w hen review ing the code!

Now , w e just have to connect up all of the output pins to our LEDs, ensuring that a resistor is placed before the LEDs as to reduce the current and that the cathodes of the LEDs go back to ground. For the sake of keeping cable density to a minimum, I placed my resistors and LEDs on a separate breadboard how ever if you're happy to use the one breadboard you can.

When placing the LEDs be sure that they are connected in order, so that QA is w ired to the first LED, and QH is w ired to the last LED, as otherw ise our code is not going to light up the LEDs in the correct order!

When you're done you should have something that looks similar to the illustration and photo below .





Now w e're ready to put some code behind it! Plug your Arduino into your computer and upload the follow ing sketch to it:

```
int latchPin = 12;
int clockPin = 11;
int dataPin = 13;
byte leds = 0;
int currentLED = 0;

void setup()
{
    pinMode(latchPin, OUTPUT);
    pinMode(dataPin, OUTPUT);
    pinMode(clockPin, OUTPUT);

    leds = 0;
}

void loop()
{
    leds = 0;

    if (currentLED == 7)
    {
        currentLED = 0;
    }
    else
    {
        currentLED++;
    }

    bitSet(leds, currentLED);

    digitalWrite(latchPin, LOW);
    shiftOut(dataPin, clockPin, LSBFIRST, leds);
    digitalWrite(latchPin, HIGH);

    delay(250);
}
```

To start with, we define the following at the top of the sketch:

- The location of the latch, clock and data pins
- A byte which will store the bits that indicate to the shift register which outputs to use
- A variable that will keep track of which LED we should be lighting up

In the setup method we simply initialise the pin modes and the leds variable.

In the loop method, we clear the bits in the leds variable at the start of every iteration so that all the bits are set to 0 as we only want to light up one LED at a time. After this we increment or reset the currentLED variable so that we are lighting up the correct LED next.

After these two operations we get to the more important part – the bit shifting. We first start by making a call to the `bitSet` method. We pass through to the bitSet method the byte that we are storing the bits in, and the currentLED variable.

This method allows us to set individual bits of the byte by specifying its position. For example, if we wanted to manually set the byte to be **10010**, we could use the following calls, as the bits we need to set to 1 are the second from the right (which is position 1, as we start at position 0), and the fifth one from the right, which is at position 4:

```
bitSet(leds, 1);
bitSet(leds, 4);
```

So, every time we increment the currentLED variable and pass it to the bitSet method, we are setting the bit to the left of the previous one to 1 every time, and thus telling the shift register to activate the output to the left of the previous one.

After setting the bit(s) we write to the latch pin in order to indicate to the shift register that we are about to send it the data. Once we have done this we make a call to the `shiftOut` method that is provided to us courtesy of Arduino. This method is designed specifically for the purpose of using shift registers, and allows us to simply shift the bits in one call. To do this we pass through the data and clock pins as the first two parameters, we then pass the LSBFIRST constant, which tells the method that the first bit should be the least significant (i.e. the right most bit) and then we pass through the byte containing the bits that we actually want to shift to the shift register.

Once we have completed the bit shifting, we write to the latch pin again (using HIGH this time) to indicate that we have sent all the data. After that write operation is complete, the matching LED will light up and then we just delay for 250 milliseconds before doing it again!

If you've completed all these steps correctly you should have something similar to that in the video below. **Note**: in the video below I used a slightly modified version of this sketch which lights up all the LEDs sequentially, the sketch above will light one at a time.

Arduino 74HC595 Shift Register Demo

Something that should be noted going forward is that this particular shift register cannot safely draw a current higher than 70 mA at a time (which is why the guide shows you how to light one LED at a time instead of all of them).

If you are going to need to draw more than this then you should look into something a bit more powerful such as the TPIC6B595. Alternatively, if you are using the shift register for just LED purposes, try some bigger resistors. If a 680 ohm resistor is placed before each LED this should be enough to keep within roughly 20-30% of the maximum current rating and still keep a relatively bright LED.

Tagged: arduino, 74HC595, expand, pins, shift, register

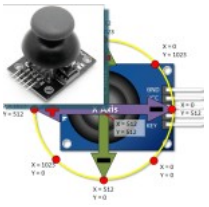
Arduino PS2 Joystick Tutorial: Keyes KY-023 Deek Robot

来源网址： http://henrysbench.capnfatz.com/henrys-bench/arduino-sensors-and-input/arduino-ps2-joystick-tutorial-keyes-ky-023-deek-robot/
作者： 王景夏

Arduino PS2 Joystick Tutorial: Keyes KY-023 Deek Robot

Contents [show]

A Versatile Input Device



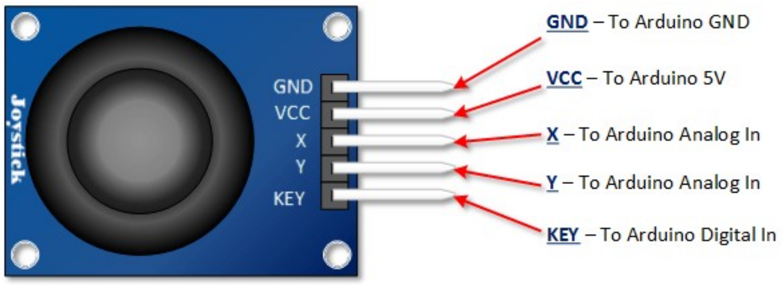
The PS2 style joystick is a thumb operated device, that when put to creative use, offers a convenient way of getting operator input. Its fundamentally consists of two potentiometers and a push button switch.

The two potentiometers indicate which direction the potentiometer is being pushed.

The switch sends a low (or ground) when the joy stick knob is pressed.

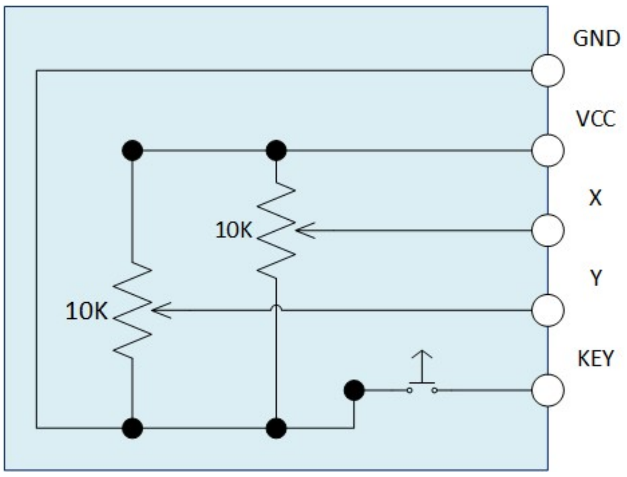
Arduino PS2 Joystick Pin Outs

This input device interfaces to your Arduino via five pins. Three of which are inputs to your Arduino, while the remaining two supply voltage and ground.



Arduino PS2 Joystick Schematic

As you can see in the schematic below, full deflection of a potentiometer in either direction will provide ground or the supply voltage as an output.

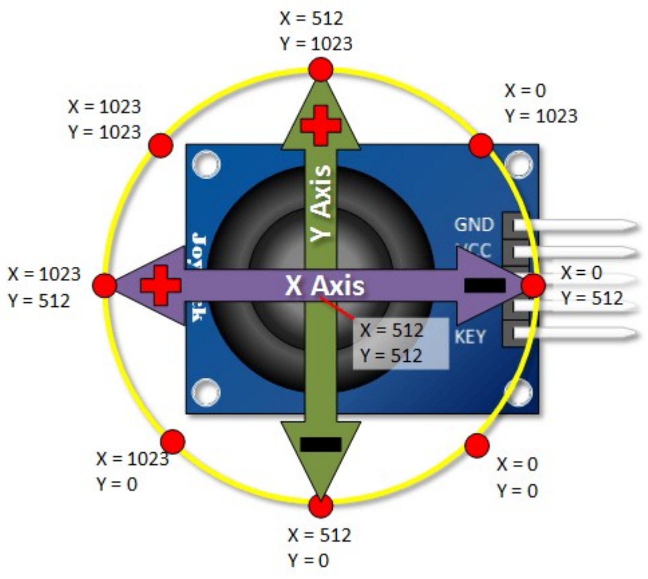


Arduino PS2 Joystick Output Orientation

In order to put this thumb control to use, you are going to want to understand which direction is X and which direction is Y. You will also need to decipher the direction it is being pushed in either the X or the Y direction.

In this tutorial we are using analog inputs to measure the joystick position. The analog inputs provided indications that range between 0 and 1023.

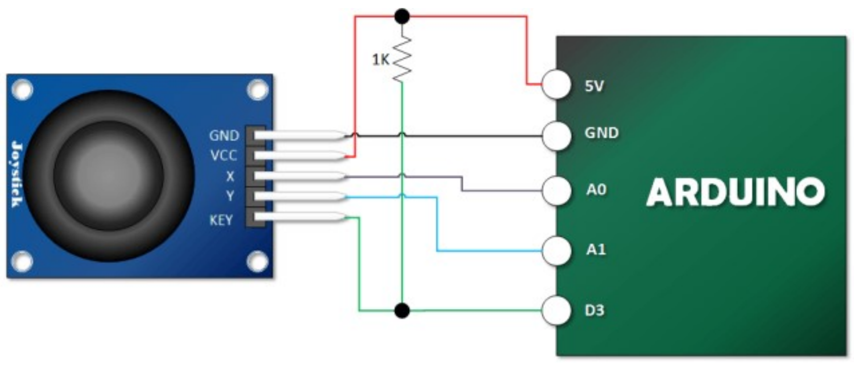
The graphic below shows the X and Y directions and also gives an indication of how the outputs will respond when the joystick is pushed in various directions. Keep in mind, the graphic you see is based on my Deek-Robot model and may in fact differ a little with yours. It that' s the case, experiment a little and draw your own sketch so that the orientations are clear.



Arduino PS2 Joystick Tutorial

Assemble the PS2 Joystick Project

Note that I use a pull up resistor between the key switch and the digital input. Once you move beyond experimentation, I highly recommend some sort of software or hardware debounce for this switch as well.



Load Your PS2 Tutorial Sketch

```
// Henry's Bench// Module KY023

int Xin= A0; // X Input Pinint Yin = A1; // Y Input Pinint KEYin = 3; // Push Button

void setup ()
{
  pinMode (KEYin, INPUT);
  Serial.begin (9600);
}
void loop ()
{
  int xVal, yVal, buttonVal;

  xVal = analogRead (Xin);
  yVal = analogRead (Yin);
  buttonVal = digitalRead (KEYin);

  Serial.print("X = ");
  Serial.println (xVal, DEC);

  Serial.print ("Y = ");
  Serial.println (yVal, DEC);

  Serial.print("Button is ");
  if (buttonVal == HIGH){
    Serial.println ("not pressed");
  }
  else{
    Serial.println ("PRESSED");
  }

  delay (500);
}
```