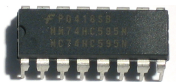# Project 15

Shift Register 8-Bit Binary Counter

Project 15 – Shift Register 8-Bit Binary Counter

Right, we are now going to delve into some pretty advanced stuff so you might want a stiff drink before going any further.

In this project we are going to use additional IC's (Integrated Circuits) in the form of Shift Registers, to enable us to drive LED's to count in Binary (we will explain what binary is soon). In this project we will drive 8 LED's independently using just 3 output pins from the Arduino.

What you will need

|	
1 x 74HC595 Shift Registers	
8 x 240Ω Resistor	
8 x Green LED	

Connect it up

Examine the diagram carefully. Connect the 3.3v to the top rail of your Breadboard and the Ground to the bottom. The chip has a small dimple on one end, this dimple goes to the left. Pin 1 is below the dimple, Pin 8 at bottom right, Pin 9 at top right and Pin 16 at top left.

You now need wires to go from the 3.3v supply to Pins 10 & 16. Also, wires from Ground to Pins 8 & 13.
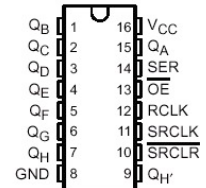
A wire goes from Digital Pin 8 to Pin 12 on the IC. Another one goes from Digital Pin 10 to Pin 14 and finally one from Digital Pin 12 to Pin 11.

The 8 LED's have a 240Ω resistor between the cathode and ground, then the anode of LED 1 goes to Pin 15. The anode of LED's 2 to 8 goes to Pins 1 to 7 on the IC.

Once you have connected everything up, have one final check your wiring is correct and the IC and LED's are the right way around. Then enter the following code. Remember, if you don't want to enter the code by hand you can download it from the website on the same page you obtained this book.
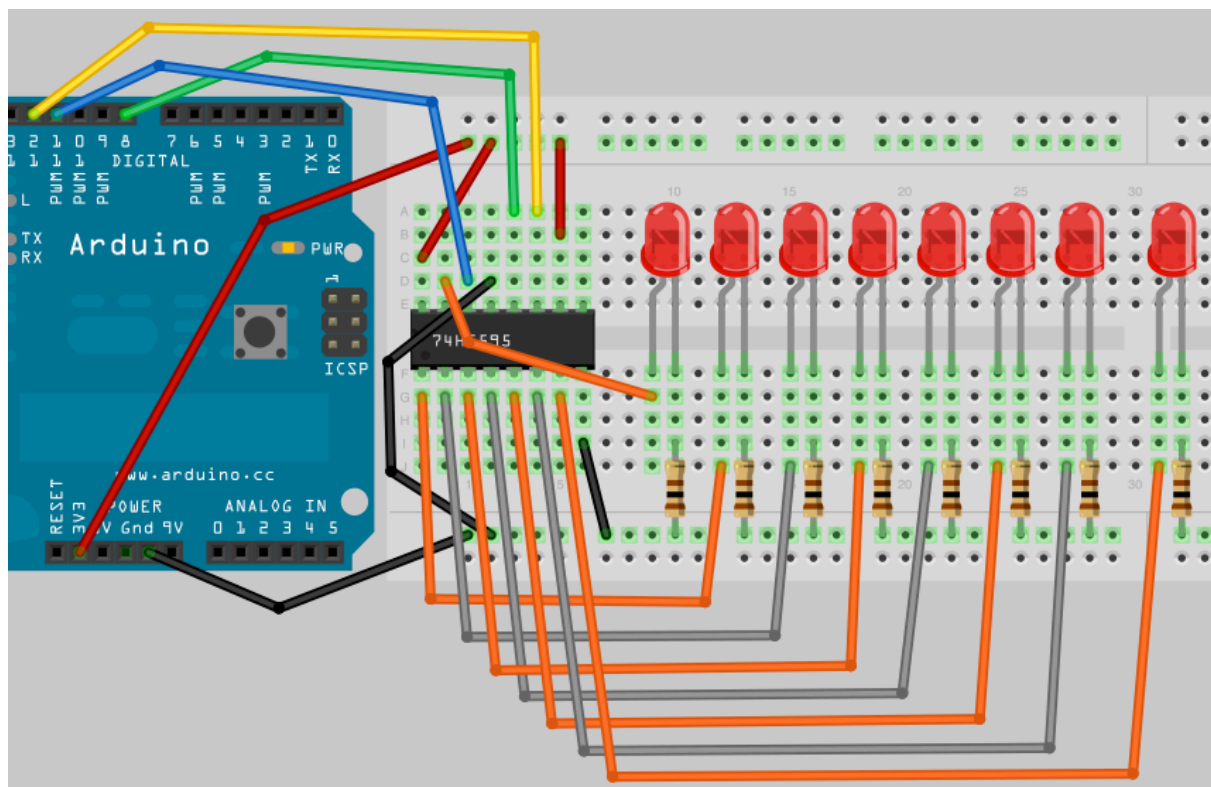
Enter the Code

Enter the following code and upload it to your Arduino. Once the code is run you will see the LED's turn on and off individually as the LED's count up in Binary from 0 to 255, then start again.

```
// Project 15

//Pin connected to Pin 12 of 74HC595 (Latch)
int latchPin = 8;
//Pin connected to Pin 11 of 74HC595 (Clock)
int clockPin = 12;
//Pin connected to Pin 14 of 74HC595 (Data)
int dataPin = 11;

void setup() {
  //set pins to output
  pinMode(latchPin, OUTPUT);
  pinMode(clockPin, OUTPUT);
  pinMode(dataPin, OUTPUT);
}

void loop() {
  //count from 0 to 255
  for (int i = 0; i < 256; i++) {
    //set latchPin low to allow data flow
    digitalWrite(latchPin, LOW);
    shiftOut(i);
    //set latchPin to high to lock and send data
    digitalWrite(latchPin, HIGH);
    delay(500);
  }
}

void shiftOut(byte dataOut) {
  // Shift out 8 bits LSB first,
  // on rising edge of clock

  boolean pinState;

   //clear shift register ready for
sending data
  digitalWrite(dataPin, LOW);
  digitalWrite(clockPin, LOW);
  // for each bit in dataOut send out
a bit
  for (int i=0; i<=7; i++)  {
   //set clockPin to LOW prior to sending bit
    digitalWrite(clockPin, LOW);

   // if the value of DataOut and (logical
AND) a bitmask
  // are true, set pinState to 1 (HIGH)
    if ( dataOut & (1<<i) ) {
      pinState = HIGH;
    }
    else {
      pinState = LOW;
    }

  //sets dataPin to HIGH or LOW depending on
pinState
    digitalWrite(dataPin, pinState);
    //send bit out on rising edge of clock
    digitalWrite(clockPin, HIGH);
  }

  //stop shifting out data
  digitalWrite(clockPin, LOW);
}
```

86

The Binary Number System

Now before we take a look at the code and the hardware for Project 15, it is time to take a look at the Binary Number System, as it is essential to understand Binary to be able to successfully program a microcontroller.

Human beings use a Base 10, or Decimal number system, because we have 10 fingers on our hands. Computers do not have fingers and so the best way for a computer to count is using it's equivalent of fingers, which is a state of either ON or OFF (1 or 0). A logic device, such as a computer, can detect if a voltage is there (1) or if it is not (0) and so uses a binary, or base 2 number system as this number system can easily be represented in an electronic circuit with a high or low voltage state.

In our number system, base 10, we have 10 digits ranging from 0 to 9. When we count to the next digit after 9 the digit resets back to zero, but a 1 is incremented to the tens column to its left. Once the tens column reaches 9, incrementing this by 1 will reset it to zero, but add 1 to the hundreds column to it's left, and so on.

```
000,001,002,003,004,005,006,007,008,009
010,011,012,013,014,015,016,017,018,019
020,021,023 ………
```

In Binary the exact same thing happens, except the highest digit is 1 so adding 1 to 1 results in the digit resetting to zero and 1 being added to the column to the left.

```
000, 001
010, 011
100, 101...
```

An 8 bit number (or a byte) is represented like this

2^7 128	2^6 64	2^5 32	2^4 16	2^3 8	2^2 4	2^1 2	2^0 1
0	1	0	0	1	0	1	1

The number above in Binary is 1001011 and in Decimal this is 75.

This is worked out like this :

1 x 1 = 1
1 x 2 = 2
1 x 8 = 8
1 x 64 = 64

Add that all together and you get 75.

Here are some other examples:

Dec	2^7 128	2^6 64	2^5 32	2^4 16	2^3 8	2^2 4	2^1 2	2^0 1
75	0	1	0	0	1	0	1	1
1	0	0	0	0	0	0	0	1
2	0	0	0	0	0	0	1	0
3	0	0	0	0	0	0	1	1
4	0	0	0	0	0	1	0	0
12	0	0	0	0	1	1	0	0
27	0	0	0	1	1	0	1	1
100	0	1	1	0	0	1	0	0
127	0	1	1	1	1	1	1	1
255	1	1	1	1	1	1	1	1

...and so on.

So now that you understand binary (or at least I hope you do) we will first take a look at the hardware, before looking at the code.

TOP TIP

You can use Google to convert between a Decimal and a Binary number and vice versa.

E.g to convert 171 Decimal to Binary type
171 in Binary
Into the Google search box returns
171 = 0b10101011

The 0b prefix shows the number is a Binary number and not a Decimal number.
So the answer is 10101011.

To convert a Binary number to decimal do the reverse. E.g. Enter
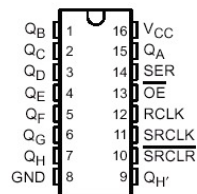***0b11001100 in Decimal***
Into the search box returns
0b11001100 = 204

Project 15 - Hardware Overview

We are going to do things the other way around for this project and take a look at the hardware before we look at the code.

We are using a Shift Register. Specifically the 74HC595 type of Shift Register. This type of Shift Register is an 8-bit serial-in, serial or parallel-out shift register with output latches. This means that you can send data in to the Shift Register in series and send it out in parallel. In series means 1 bit at a time. Parallel means lots of bits (in this case 8) at a time. So you give the Shift Register data (in the form of 1's and 0's) one bit at a time, then send out 8 bits all at the exact same time. Each bit is shunted along as the next bit is entered. If a 9th bit is entered before the Latch is set to HIGH then the first bit entered will be shunted off the end of the row and be lost forever.

Shift Registers are usually used for serial to parallel data conversion. In our case, as the data that is output is 1's and 0's (or 0v and 3.3v) we can use it to turn on and off a bank of 8 LED's.

```
        ┌───∪───┐
Q_B  [ 1      16 ] V_CC
Q_C  [ 2      15 ] Q_A
Q_D  [ 3      14 ] SER
Q_E  [ 4      13 ] OE
Q_F  [ 5      12 ] RCLK
Q_G  [ 6      11 ] SRCLK
Q_H  [ 7      10 ] SRCLR
GND  [ 8       9 ] Q_H'
        └───────┘
```

The Shift Register, for this project, requires only 3 inputs from the Arduino. The outputs of the Arduino and the inputs of the 595 are as follows:

Arduino Pin	595 Pin	Description
8	12	Storage Register Clock Input
11	14	Serial Data Input
12	11	Shift Register Clock Input

We are going to refer to Pin 12 as the Clock Pin, Pin 14 as the Data Pin and Pin 11 as the Latch Pin.

Imagine the Latch as a gate that will allow data to escape from the 595. When the gate is lowered (LOW) the data in the 595 cannot get out, but data can be entered. When the gate is raised (HIGH) data can no longer be entered, but the data in the SHift Register is

released to the 8 Pins (QA-QH). The Clock is simply a pulse of 0's and 1's and and the Data Pin is where we send data from the Arduino the the 595.
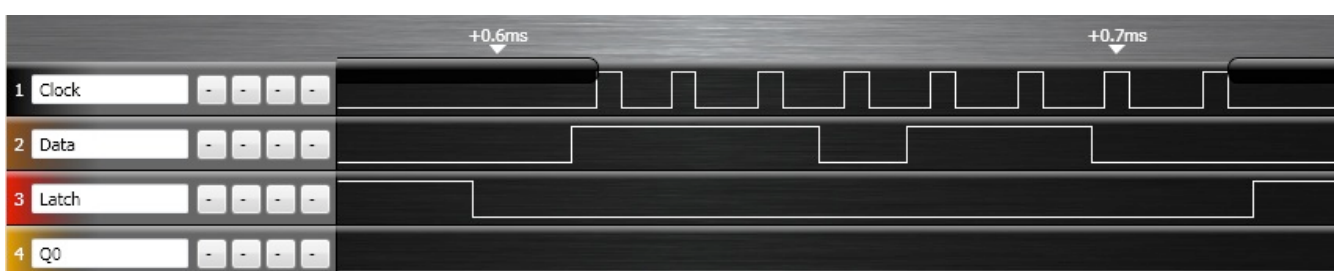
To use the Shift Register the Latch Pin and Clock Pin must be set to LOW. The Latch Pin will remain at LOW until all 8 bits have been set. This allows data to be entered into the Storage Register (the storage register is simply a place inside the IC for storing a 1 or a 0). We then present either a HIGH or LOW signal at the Data Pin and then set the Clock Pin to HIGH. By setting the Clock Pin to HIGH this stores the data presented at the Data Pin into the Storage Register. Once this is done we set the Clock to LOW again, then present the next bit of data at the Data Pin. Once we have done this 8 times, we have sent a full 8 bit number into the 595. The Latch Pin is now raised which transfers the data from the Storage Register into the Shift Register and outputs it from QA to QH (Pin 15, 1 to 7).

I have connected a Logic Analyser (a device that lets you see the 1's and 0's coming out of a digital device)t o my 595 whilst this program is running and the image at the bottom of the page shows the output.

The sequence of events here is:

| Pin	State	Description
Latch	LOW	Latch lowered to allow data to be entered
Data	HIGH	First bit of data (1)
Clock	HIGH	Clock goes HIGH. Data stored.
Clock	LOW	Ready for next Bit. Prevent any new data.
Data	HIGH	2nd bit of data (1)
Clock	HIGH	2nd bit stored
...	...	…
Data	LOW	8th bit of data (0)
Clock	HIGH	Store the data
Clock	LOW	Prevent any new data being stored
Latch	HIGH	Send 8 bits out in parallel

In the image below, you can see that the binary number 00110111 (reading from right to left) or Decimal 55 has been sent to the chip.

So to summarise the use of a single Shift Register in this project, we have 8 LED's attached to the 8 outputs of the Register. The Latch is set to LOW to enable data entry. Data is sent to the Data Pin, one bit at a time, the CLock Pin is set to HIGH to store that data, then back down to low ready for the next bit. After all 8 bits have been entered, the latch is set to HIGH which prevents further data entry and sets the 8 output pins to either High (3.3v or LOW (0 volts) depending on the state of the Register.

If you want to read up more about the shift register you have in your kit, then take a look at the serial number on the IC (e.g. 74HC595N or SN74HC595N, etc.) and enter that into Google. You can then find the specific datasheet for the IC and read more about it.

I'm a huge fan of the 595 chip. It is very versatile and can of course increase the number of digital output pins that the Arduino has. The standard Arduino has 19 Digital Outputs (the 6 Analog Pins can also be used as Digital Pins numbered 14 to 19). Using 8-bit Shift Registers you can expand that to 49 (6 x 595's plus one spare pin left over). They also operate very fast, typically at 100MHz. Meaning you can send data out at approx. 100 million times per second if you wanted to. This means you can also send PWM signals via software to the IC's and enable brightness control of the LED's too.
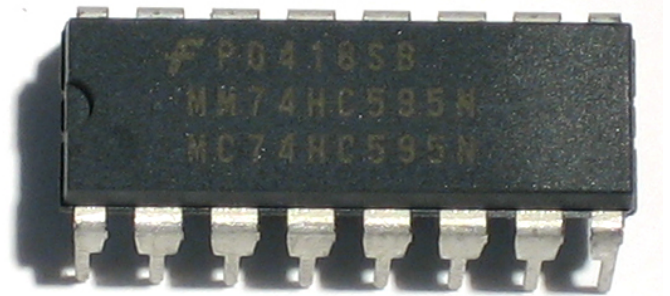
As the outputs are simply ON's and OFF's of an output voltage, they can also be used to switch other low powered (or even high powered devices with the use of transistors or relays) devices on and off or to send data to devices (e.g. An old dot matrix printer or other serial device).

All of the 595 Shift Registers from any manufacturer are just about identical to each other. There are also larger Shift Registers with 16 outputs or higher. Some IC's advertised as LED Driver Chips are, when you examine the datasheet, simply larger Shift Registers (e.g. The M5450 and M5451 from STMicroelectronics).

Project 15 – Code Overview

The code for Project 15 looks pretty daunting at first look. But when you break it down into it's component parts.

First, 3 variables are initialised for the 3 pins we are going to use.

```
int latchPin = 8;
int clockPin = 12;
int dataPin = 11;
```

Then, in setup, the pins are all set to Outputs.

```
  pinMode(latchPin, OUTPUT);
  pinMode(clockPin, OUTPUT);
  pinMode(dataPin, OUTPUT);
```

The main loop simply runs a for loop counting from 0 to 255. On each iteration of the loop the latchPin is set to LOW to enable data entry, then the function called shiftOut is called, passing the value of i in the for loop to the function. Then the latchpin is set to HIGH, preventing further data entry and setting the outputs from the 8 pins. Finally there is a delay of half a second before the next iteration of the loop commences.

```
void loop() {
  //count from 0 to 255
  for (int i = 0; i < 256; i++) {
    //set latchPin low to allow data flow
    digitalWrite(latchPin, LOW);
    shiftOut(i);
     //set latchPin to high to lock and send
data
    digitalWrite(latchPin, HIGH);
    delay(500);
  }
}
```

The shiftOut function receives as a parameter a Byte (8 bit number), which will be our number between 0 and 255. We have chosen a Byte for this usage as it is exactly 8 bits in length and we need to send only 8 bits out to the Shift Register.

```
void shiftOut(byte dataOut) {
```

Then a boolean variable called pinState is initialised. This will store the state we wish the relevant pin to be in when the data is sent out (1 or 0).

```
  boolean pinState;
```

The Data and Clock pins are set to LOW to reset the data and clock lines ready for fresh data to be sent.

```
  digitalWrite(dataPin, LOW);
  digitalWrite(clockPin, LOW);
```

After this, we are ready to send the 8 bits in series to the 595 one bit at a time.

A for loop that iterates 8 times is set up.

```
    for (int i=0; i<=7; i++)  {
```

The clock pin is set low prior to sending a Data bit.

```
      digitalWrite(clockPin, LOW);
```

Now an if/else statement determines if the pinState variable should be set to a 1 or a 0.

```
    if ( dataOut & (1<<i) ) {
      pinState = HIGH;
    }
    else {
      pinState = LOW;
    }
```

The condition for the if statement is:
```
 dataOut & (1<<i).
```

This is an example of what is called a 'bitmask' and we are now using Bitwise Operators. These are logical operators similar to the Boolean Operators we used in previous projects. However, the Bitwise Operators act on number at the bit level.

In this case we are using the Bitwise and (&) operator to carry out a logical operation on two numbers. The first number is dataOut and the second is the result of (1<<i). Before we go any further let's take a look at the Bitwise Operators.

Bitwise Operators

The Bitwise Operators perform calculations at the bit level on variables. There are 6 common Bitwise Operators and these are:

```
&       Bitwise and
|       Bitwise or
^       Bitwise xor
~       Bitwise not
<<      Bitshift left
>>      Bitshift right
```

Bitwise Operators can only be used between integers. Each operator performs a calculation based on a set of logic rules. Let us take a close look at the Bitwsie AND (&) Operator.

Bitwise AND (&)

The Bitwise AND operator act according to this rule:-

If both inputs are 1, the resulting outputs are 1, otherwise the output is 0.

Another way of looking at this is:

```
0 0 1 1       Operand1
0 1 0 1       Operand2
-------
0 0 0 1       (Operand1 & Operand2)
```

A type int is a 16-bit value, so using & between two int expressions causes 16 simultaneous AND operations to occur. In a section of code like this:

```
int x = 77;   //binary: 0000000001001101
int y = 121;  //binary: 0000000001111001
int z = x & y;//result: 0000000001001001
```

Or in this case 77 & 121 = 73

The remaining operators are:

Bitwise OR (I)

If either or both of the inputs is 1, the result is 1, otherwise it is 0.

```
0 0 1 1       Operand1
0 1 0 1       Operand2
-------
0 1 1 1       (Operand1 | Operand2)
```

Bitwise XOR (^)

If only 1 of the inputs is 1, then the output is 1. If both inputs are 1, then the output 0.

```
0 0 1 1       Operand1
0 1 0 1       Operand2
-------
0 1 1 0       (Operand1 ^ Operand2)
```

Bitwise NOT (~)

The Bitwise NOT Operator is applied to a single operand to its right.

The output becomes the opposite of the input.

```
0 0 1 1       Operand1
-------
1 1 0 0       ~Operand1
```

Bitshift Left (<<), Bitshift Right (>>)

The Bitshift operators move all of the bits in the integer to the left or right the number of bits specified by the right operand.

variable << number_of_bits

E.g.

```
byte x=9 ;  // binary: 00001001
byte y=x<<3; //binary: 01001000 (or 72 dec)
```

Any bits shifted off the end of the row are lost forever. You can use the left bitshift to multiply a number by powers of 2 and the right bitshift to divide by powers of 2 (work it out).

Now that we have taken a look at the Bitshift Operators let's return to our code.

Project 15 - Code Overview (continued)

The condition of the if/else statement was

```
dataOut & (1<<i)
```

And we now know this is a Bitwise AND (&) operation. The right hand operand inside the parenthesis is a left bitshift operation. This is a 'bitmask'. The 74HC595 will only accept data one bit at a time. We therefore need to convert the 8 bit number in dataOut into a single bit number representing each of the 8 bits in turn. The bitmask allows us to ensure that the pinState variable is set to either a 1 or a 0 depending on what the result of the bitmask calculation is. The right hand operand is the number 1 bit shifted i number of times. As the for loop makes the value of i go from 0 to 7 we can see that 1 bitshifted i times, each time through the loop, will result in these binary numbers:

Value of I	Result of (1<<i) in Binary
0	00000001
1	00000010
2	00000100
3	00001000
4	00010000
5	00100000
6	01000000
7	10000000

So you can see that the 1 moves from right to left as a result of this operation.

Now the & operator's rules state that

If both inputs are 1, the resulting outputs are 1, otherwise the output is 0.

So, the condition of

```
dataOut & (1<<i)
```

will result in a 1 if the corresponding bit in the same place as the bitmask is a 1, otherwise it will be a zero. For example, if the value of dataOut was Decimal 139 or 10001011 binary. Then each iteration through the loop will result in

Value of I	Result of b10001011(1<<i) in Binary
0	00000001
1	00000010
2	00000000
3	00001000
4	00000000

Value of I	Result of b10001011(1<<i) in Binary
5	00000000
6	00000000
7	10000000

So every time there is a 1 in the I position (reading from right to left) the value comes out at higher than 1 (or TRUE) and every time there is a 0 in the I position, the value comes out at 0 (or FALSE).

The if condition will therefore carry out its code in the block if the value is higher than 0 (or in other words if the bit in that position is a 1) or 'else' (if the bit inthat position is a 0) it will carry out the code in the else block.

So looking at the if/else statement once more

```
if ( dataOut & (1<<i) ) {
    pinState = HIGH;
}
else {
    pinState = LOW;
}
```

And cross referenciong this with the truth table above, we can see that for every bit in the value of dataOut that has the value of 1 that pinState will be set to HIGH and for every value of 0 it will be set to LOW.

The next piece of code writes either a HIGH or LOW state to the Data Pin and then sets the Clock Pin to HIGH to write that bit into the storage register.

```
digitalWrite(dataPin, pinState);
digitalWrite(clockPin, HIGH);
```

Finally the Clock Pin is set to low to ensure no further bit writes.

```
digitalWrite(clockPin, LOW);
```

So, in simple terms, this section of code looks at each of the 8 bits of the value in dataOut one by one and sets the data pin to HIGH or LOW accordingly, then writes that value into the storage register.

This is simply sending the 8 bit number out to the 595 one bit at a time and then the main loop sets the Latch Pin to HIGH to send out those 8 bits simultaneously to Pins 15 and 1 to 7 (QA to QH) of the Shift Register, thus making our 8 LED's show a visual representation of the binary number stored in the Shift Register.