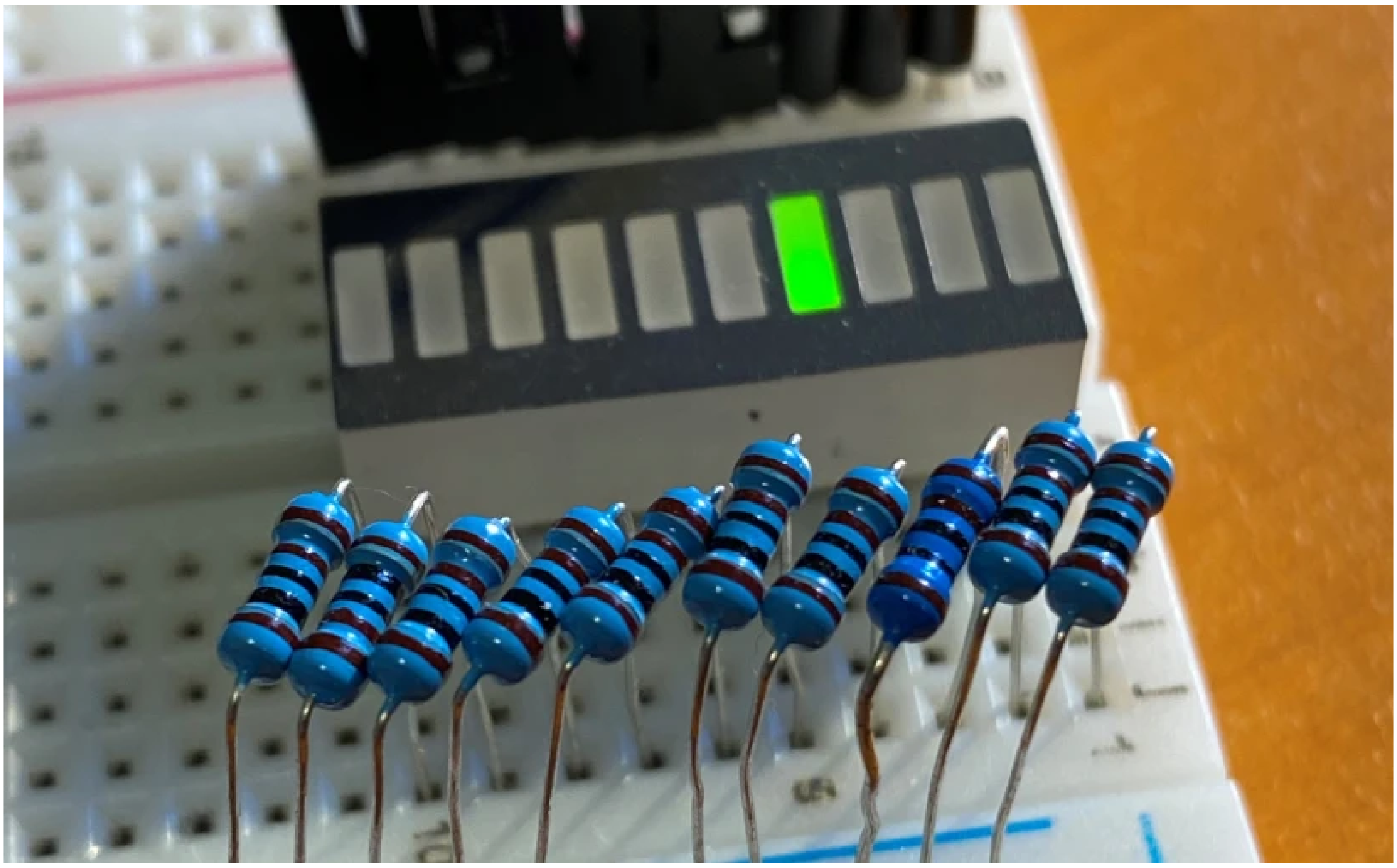


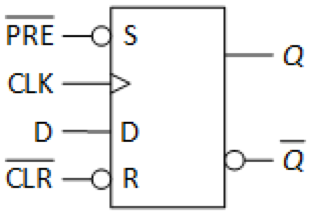
Using Shift Registers to Increase Digital Outputs

The Arduino Uno is limited to fourteen digital input/output pins and six analog input pins. The analog input pins can also be used for digital input/output thus providing a total of 20 digital input/output pins. In the four digit seven-segment display developed in blog post Driving Seven Segment Displays, twelve digital input/output pins were used, seriously limiting the amount of input/output real estate left. This would be especially true if we were to increase the number of displayed digits and if we needed to do any useful interaction with the environment through sensors. What if we needed tens or even hundreds of digital outputs from a single Arduino Uno? It can be done using digital electronic integrated circuits called shift registers along with a technique called serial communication.

I have used serial communication throughout my blog posts. We have used it to send and receive Morse code in the Morse Code Reader and Morse Code Generator posts. We have also used it to communicate with the temperature and humidity sensor DHT22 in the Sensing Temperature and Humidity post. In this post, we will use the SN74HC595 8-bit shift registers with 3-State output registers integrated circuits to convert serial signals into a collection digital output values. Using only three of the Arduino Uno's digital I/O ports we will send digital values to the shift register integrated circuits and we will use the shift register outputs to drive a ten-LED bar graph.

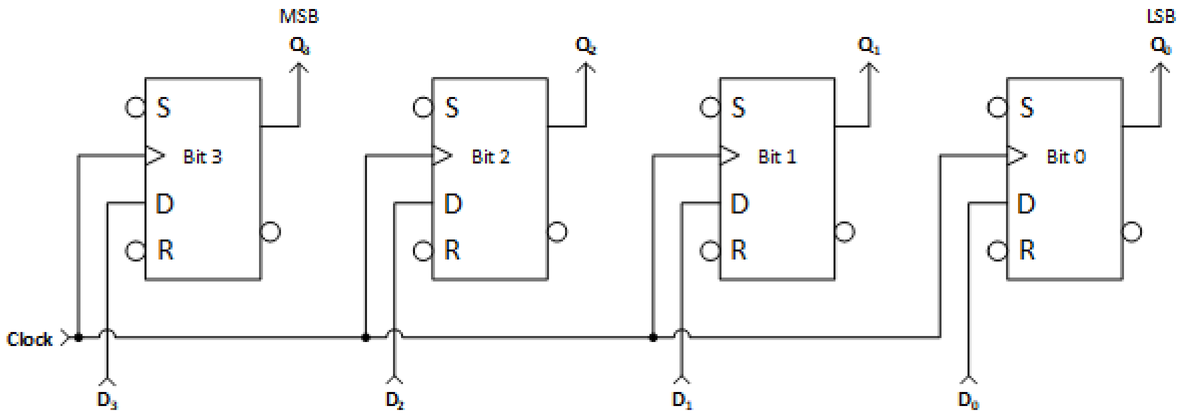
Registers

In digital electronics, a register is a collection of devices, also known as flip-flops, each capable of holding a single bit of information. Effectively, a flip-flop is a single bit of memory. The following diagram depicts a D flip-flop, a device capable of remembering the value of a digital signal, D, when another signal, CLK, goes from a low to a high value, from 0 to 1. The device has two outputs, Q and $\overline{Q}$, that provide the device's remembered value and its inverse respectively. The device has two other inputs, $\overline{PRE}$ and CLR. The $\overline{PRE}$ input sets the flip-flop value to a high value, 1, when a low pulse is applied to the input. The $\overline{CLR}$ input resets the flip-flop value to a low value, 0, when a low pulse is applied to the input.



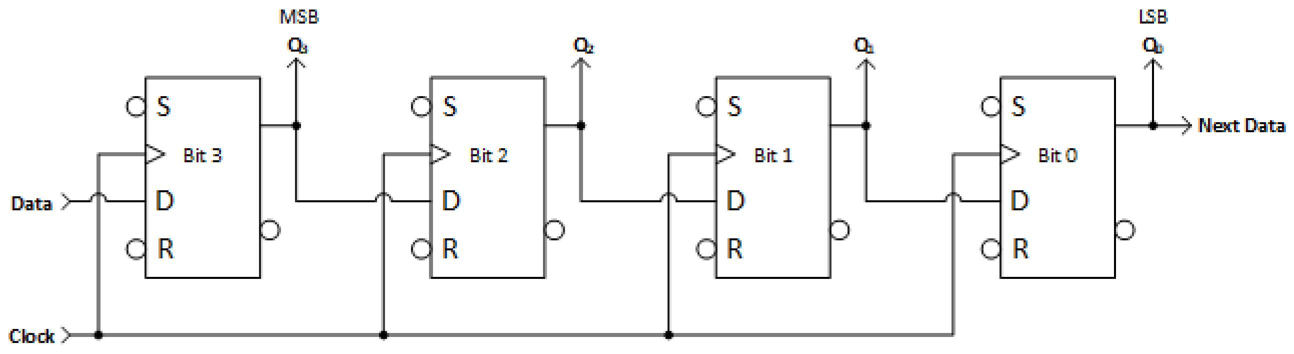
The previous diagram follows the IEEE Standard 91-1984 which defines how to represent digital electronic components. The device is represented by a box. All input lines are drawn at the left of the box while all outputs are drawn at the right. Lines connected directly to the device represent signals that are active when the value of the signal is high, 1. Lines connected to a small circle attached to the device represent signals that are active when the signal is low, 0. Lines connected to the device through a small triangle represent signals that are active when the value switches from low to high, 0 to 1.

A register, as depicted in the following drawing, is made of several flip-flops, one for each bit of information to be stored. Data is presented at the D input of each flip-flop and the state of each flip-flop is set when the clock signal, attached to the clock input of each flip-flop, is toggled from a low to a high value, 0 to 1. The data stored in the register in the form of bits of information is made available at the output of the flip-flops making up the register. In the diagram, four flip-flops store four bits of information depicted from right to left, least significant bit at the rightmost position and most significant bit at the leftmost position.



Serial to Parallel Shift Registers

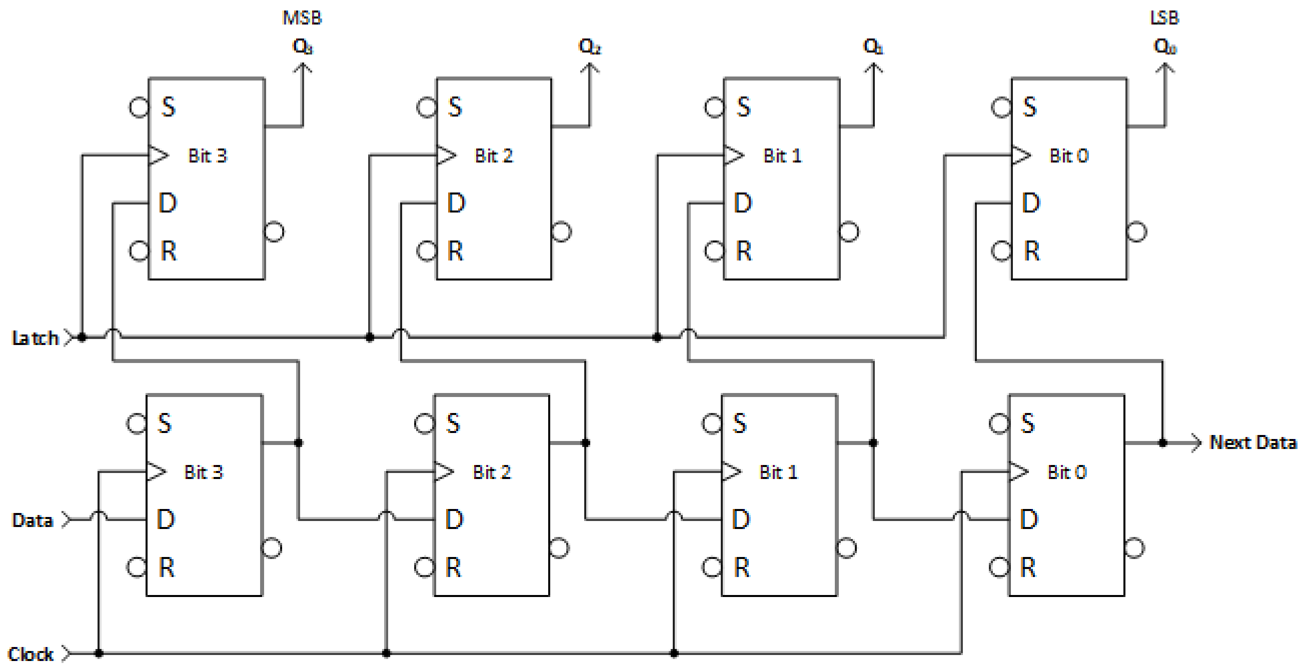
a serial to parallel shift register is a register that converts a serial train of bits into a parallel representation. The diagram below depicts the connectivity required to transform a register into a serial to parallel shift register. There are two inputs to the circuit: the data and clock signals. The data signal is connected to the most significant flip-flop's data input; each flip-flop output is connected to the next less significant flip-flop data input; and the clock signal is connected to every flip-flop clock inputs. At each clock toggle from low to high, flip-flops take the value or the previous more significant flip-flop and the most significant flip-flop takes the value of the data signal. All flip-flop outputs form the parallel representation of the successive serial values of the data signal. By presenting values, a bit at a time, least significant bit first, on the data signal, clocking in each new value, we can convert the serial data signal to a parallel representation of the signal.



The values stored in each flip-flop of the serial to parallel shift register becomes the representation of the serial signal in the end. However, while the bits are being shifted, the flip-flop outputs will contain intermediate values until we finish clocking in all the values. A way to solve this problem is to put a register to store the output of the serial to parallel shift register after all bits have been shifted.

Latched Shift Registers

The following diagram depicts a digital circuit comprised of two registers, a serial to parallel shift register at the bottom and a latch register at the top. As described in the above paragraphs, the serial to parallel shift register converts the serial data signal into its parallel representation by clocking in each bit in turn; when the shifting is done, we can store the values at the output of each serial to parallel shift register flip-flop in the latch register by toggling the latch signal from a low to high value.

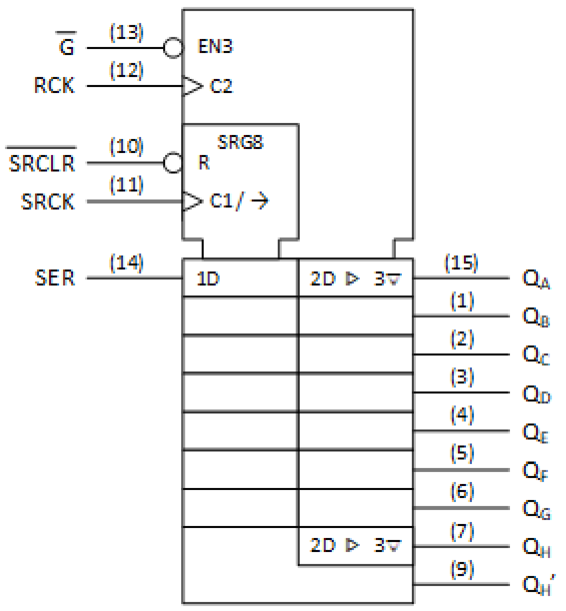


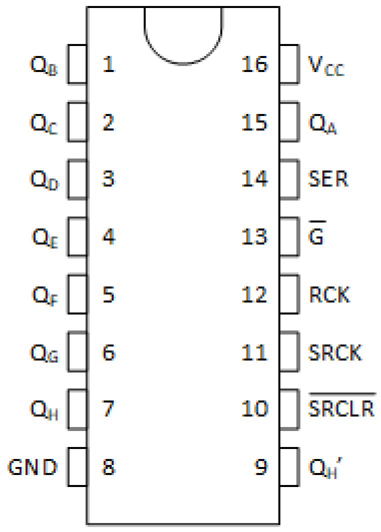
The above diagram represents a four-bit latched shift register. The circuit can be extended by repeating each serial to parallel and latch register stages to the right of the circuit by connecting the clock signal to the serial to parallel flip-flop's clock; connecting the output of the previous less significant flip-flop, called "Next Data" in the diagram, to the data input of the new serial to parallel flip-flop; by connecting the output of the new serial to parallel flip-flop to the data input of the new latch register flip-flop data input; and finally by connecting the latch signal to the clock input of the new latch register flip-flop.

The number of bits that can be supported is almost infinite, only limited by the electrical characteristics of the digital integrated circuits used to implement the circuit and the amount of time required to shift each bit in position. The integrated circuit that will be used to demonstrate how to increase the output capacity of the Arduino Uno is the SN74HC595N 8-bit shift registers with 3-State output registers digital integrated circuit. Internally, the integrated circuit is similar to the digital circuit shown above but has eight bits instead of four. Moreover, the outputs can be put in a high-impedance (high resistance) state, making it behave as if it was not connected. This latter functionality will not be used in this demonstration.

The SN74HC595 Shift Register with Latch

The diagram below is a depiction of the SN74HC595 digital integrated circuit. It describes the integrated circuit functionality on the left, and its pinout on the right. The SN74HC595 integrated circuit is housed in a 16 pin dual in-line package (DIP). The integrated circuit's pins 8 and 16, not shown on the functional diagram, are used to power the integrated circuit. Pin 8, GND, is to be connected to ground while pin 16, V_{CC}, is to be connected to power, 5 volts. In all of the following text, a `LOW` value or 0 is represented by a signal at the same potential as the ground, 0 volt, and a `HIGH` value or 1 is represented by a signal at the same potential as V_{CC}, 5 volts.





The functionality diagram follows the IEEE Standard 91-1984. The two top blocks notched at the bottom represent the integrated circuit's common control elements. The bottom blocks represent each of the eight serial register flip-flops and the output register flip-flops feeding each other in two stages, from left two right. The topmost serial register is fed external data from the serial data line through the pin labelled SER. The bottommost serial register flip-flop output is directly attached to pin Q_H' which can be used to feed another serial register's input, allowing the extension of the shift register to many more stages. Each output register flip-flop is attached to pins Q_A through Q_H giving access to each of the flip-flop's value.

There are two common control elements, one for the collection of serial register flip-flops and the other for the collection of output register flip-flops. The first, smaller, common control element has two inputs. The SRCLR input pin is used to reset to a `LOW` value, 0, all shift register flip-flops when its value is set to `LOW` for a minimum of 20 ns. The SRCK input pin is the shift register flip-flops' common clock that shifts in the value of the SER pin and shifts the shift register flip-flop values when the pin is toggled from a `LOW` to `HIGH` value, from 0 to 1. The second common control element also has two inputs. The $\overline{G}$ input pin is used to enable the integrated circuit's output register flip-flops' output when its value is `LOW` or 0 and to leave the output pins in a high impedance state when `HIGH` or 1. The RCK input pin is the output register flip-flops' common clock that latches the shift register flip-flop values into the output register flip-flops when the pin is toggled from a `LOW` to `HIGH` value.

In order to use the integrated circuit properly, we must understand its most important electrical and timing characteristics. All inputs draw a maximum of 1 µA each. The integrated circuit's output register's outputs, Q_A to Q_H can source or sink 6 mA. The SRCK and RCK shift register and output register clocks must remain `HIGH` for at least 20 ns and `LOW` for at least 20 ns, allowing for a maximum clock frequency of 25 MHz. The $\overline{\text{SRCLR}}$ input pin must remain `LOW` for at least 20 ns for the serial register flip-flops to be reset. The SER, serial data input, pin must be stable at least 25 ns before the SRCK pin can be toggled from `LOW` to `HIGH` to latch the input value.

From the integrated circuit's characteristics, we can say that an Arduino Uno digital output, which can source or sink 20 mA, can theoretically drive the inputs of up to 20 000 SN74HC595 integrated circuits if we ignore other electrical constraints. There is more to consider than the current required to drive clock inputs. We have to take into account wire lengths, the current required to drive integrated circuit outputs and integrated circuit layout. I would recommend not to daisy chain, that is serially connect clock inputs one after another, more than twenty integrated circuits without carefully considering the design of integrated circuit placement and wiring layout. As for the timing characteristics, the maximum speed at which we can toggle digital outputs on the Arduino, measured with an oscilloscope while executing two `digitalWrite()` calls is 3.5 µs, which is well above the minimum time of 20 ns required by the SN74HC595 clock inputs.

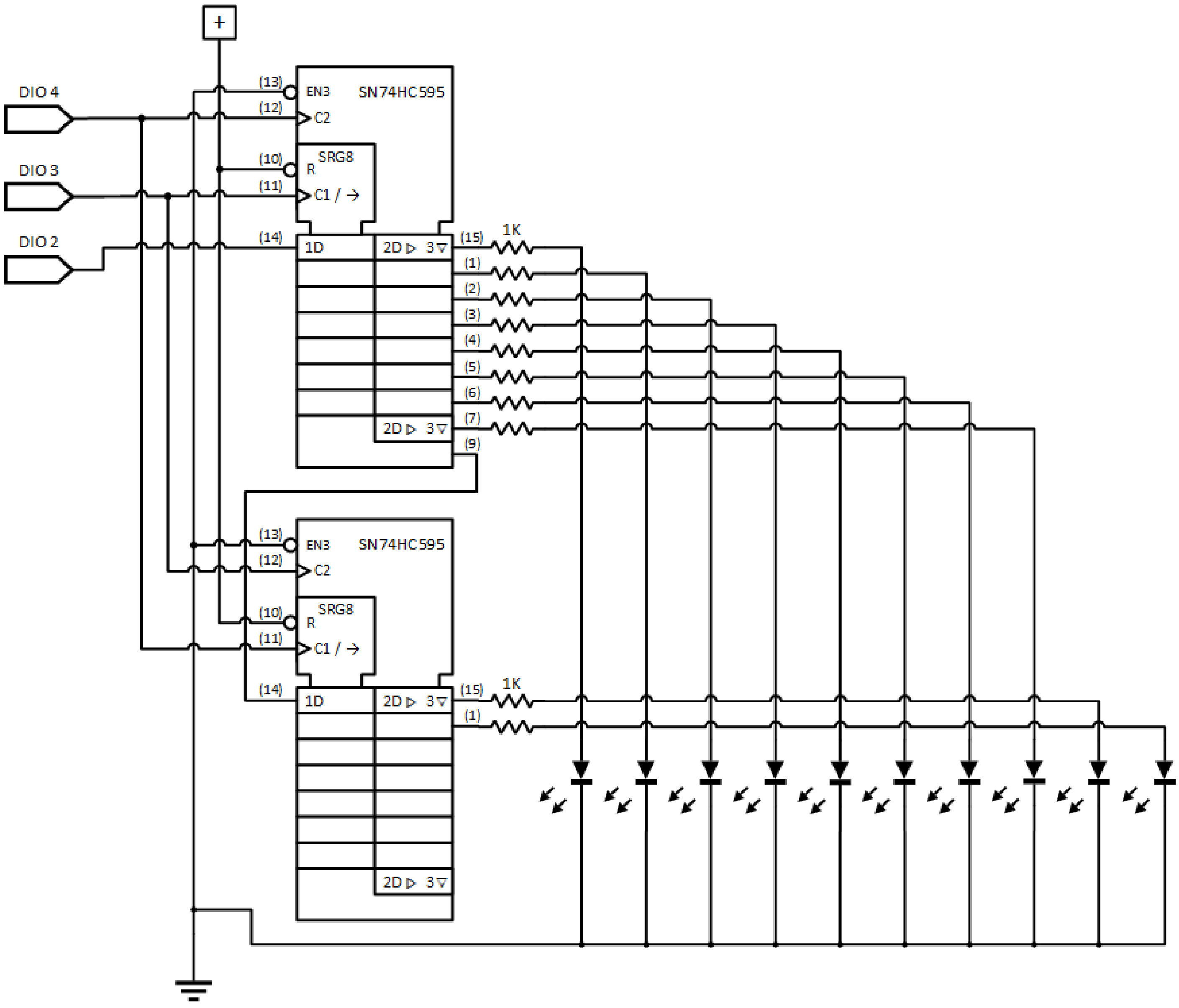
Creating 10 Digital Outputs from 3 Digital Outputs

We will use two SN74HC595 digital integrated circuits to drive a ten LED bar graph display from three Arduino digital outputs. In the following paragraphs we will discuss the circuit used to control the bar graph display using an Arduino C++ program. The program can be found on GitHub here.

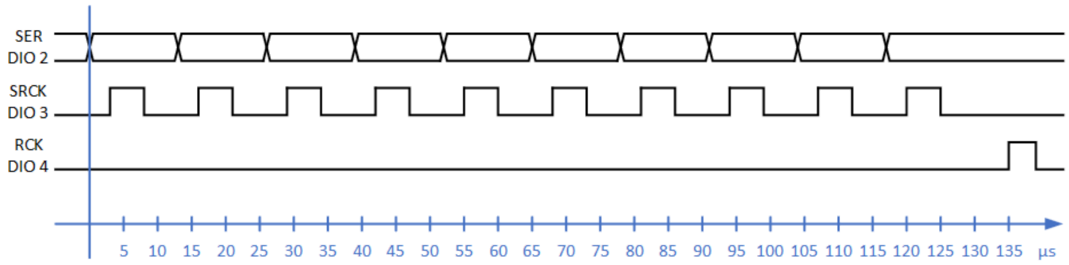
The Electronics

The following diagram depicts the digital circuit we will use to demonstrate how to drive ten bar graph LEDs from three Arduino digital outputs. In the diagram, Arduino's digital port 3 is connected to both SN74HC595 integrated circuits' serial clocks, pin 11 of each integrated circuit. Arduino's digital port 4 is connected to both SN74HC595 integrated circuits' register latch clocks, pin 12 of each integrated circuit. Arduino's digital port 2 is connected to serial data input, pin 14, of the SN74HC595 integrated circuit that is to drive the eight most significant, leftmost, LEDs of the bar graph. The serial output of that integrated circuit, pin 9, is connected to the serial data input, pin 14, of the SN74HC595 integrated circuit that is to drive the two least significant, rightmost, LEDs of the bar graph.

As shown on the diagram, the eight outputs of the topmost SN74HC595 integrated circuit are connected to the eight most significant LEDs' anode of the bar graph through 1K resistors. The two most significant outputs of the bottom SN74HC595 integrated circuit are connected to the two least significant LEDs' anode of the bar graph display through 1K resistors. All bar graph display LEDs' cathodes are connected to ground.



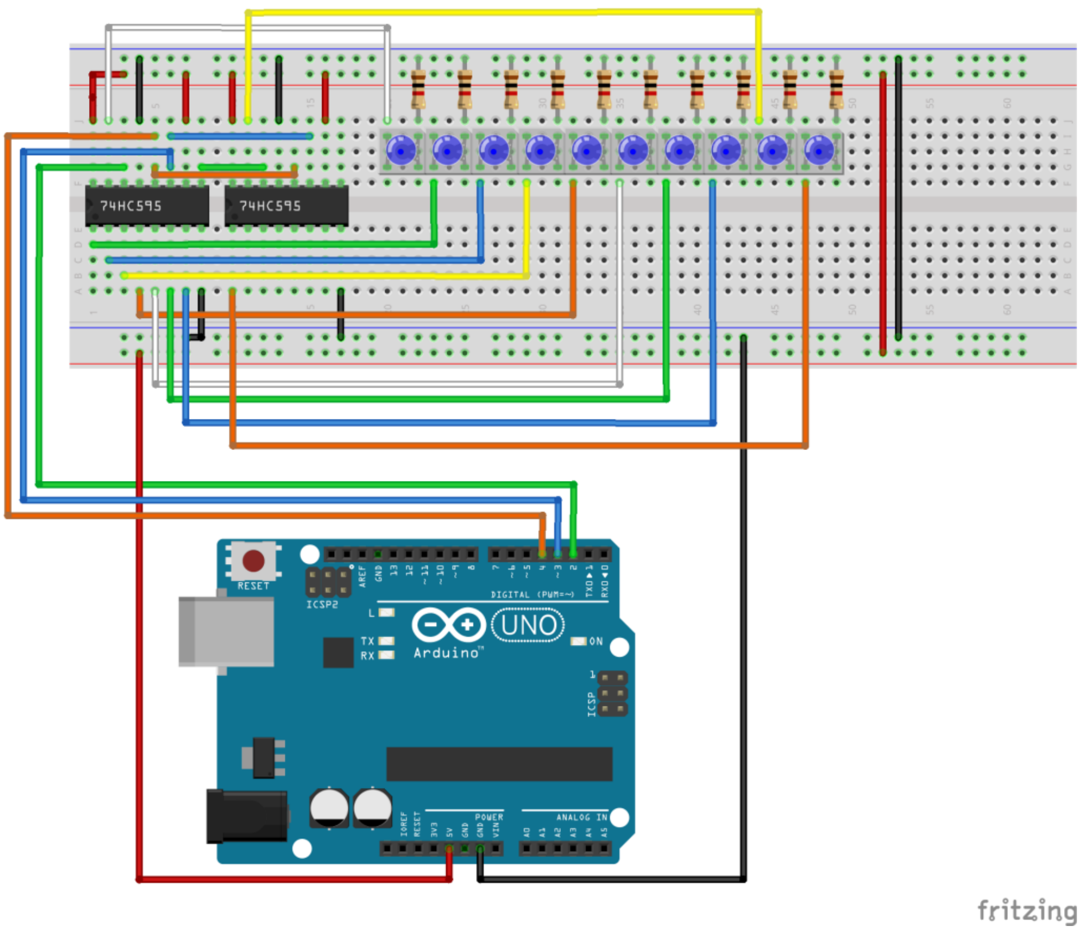
The reset pins, pin 10, of each SN74HC595 integrated circuit are connected directly to 5V, preventing the serial shift registers from resetting. The enable pins, pin 13, of each SN74HC595 integrated circuit are connected to ground thus always enabling each latched register’s output. The following timing diagram shows the digital values applied to the shift registers with latch through the Arduino digital ports to output ten digital values to be displayed on the ten LED bar graph display. The top signal in the diagram represents the serial data output by the Arduino at digital port 2. Since the data can be either a 1 or 0, a `HIGH` or a `LOW`, we show both with a crossover at the point the value is set. The second line represents the serial register clock output by the Arduino at digital port 3. The third line represents the latched output clock output by the Arduino at digital port 4.



In order to send ten binary values to the bar graph LEDs, an Arduino program must send ten bits, least significant bit first. So, the program sets the value of the least significant bit at digital port 2. The program then toggles the serial clock signal at digital port 3 from `LOW` to `HIGH`, then `HIGH` to `LOW`. These two steps are repeated for each bit of information until the most significant bit is processed. Finally, the program toggles the latch clock signal at digital port 4 so that all values that were sent serially are provided to all LEDs in the bar graph at once.

Breadboarding the Circuit

The following picture depicts how to connect the different parts using a solderless breadboard, jumper wires, SN74HC595 digital integrated circuits, 1 KΩ resistors, LEDs. and an Arduino Uno micro-controller. In the schematic, I used discrete LEDs instead of a 10 LED bar graph, but the wiring would be identical.



The Program

The program used to demonstrate the use of the latched shift register is made up of the Arduino sketch, `UsingShiftRegistersToIncreaseDigitalOutputs.ino`, containing the `setup()` and `loop()` functions. There is also a class, `LatchedShiftRegisterChannel`, that embodies the functionality of the serially connected latched shift registers. The class is defined in the `LatchedShiftRegisterChannel.h` header file and the class methods are implemented in `LatchedShiftRegisterChannel.cpp`. We described classes in a previous post, Programming with Class. In the next paragraphs we will see how the `LatchedShiftRegisterChannel` class is implemented and how the main program uses this class to output values to ten LEDs using three digital output ports.

LatchedShiftRegisterChannel.h

The `LatchedShiftRegisterChannel.h` header file defines the `LatchedShiftRegisterChannel` class. In the header file, before the class definition, we declare a constant for the maximum number of digital outputs that are supported by the class, `MAXIMUM_SERIAL_BITS`. We then declare a structure, `SerialCommunicationPins`, to store the Arduino output ports associated with the serial clock, data, and latch clock pins that are used to drive the shift register integrated circuits. Follows the `LatchedShiftRegisterChannel` class declaration. There are four public methods. `~LatchedShiftRegisterChannel()` is the class destructor called when an instance of the class goes out of scope or gets deleted. The `LatchedShiftRegisterChannel()` constructor with two arguments is the only constructor supported. It takes a `SerialCommunicationPins` structure as input as well as the maximum number bits, or digital output ports, that can be driven by the shift registers in the circuit. Each SN74HC595 integrated circuits can drive 8 bits each, hence two shift register integrated circuits can drive 16 bits. The `transmit()` method is used to output bits to the digital output ports driven by the shift registers. Finally, the `setNumberOfActiveDigitalOutputs()` method sets the actual number of digital outputs to drive. It is used if the value is different than the number of shift register bits that was set in the constructor.

```
#if !defined(LATCHEDSHIFTREGISTERCHANNEL_HEADER)
#define LATCHEDSHIFTREGISTERCHANNEL_HEADER
#include "Arduino.h"

const int MAXIMUM_SERIAL_BITS = 1024;

struct SerialCommunicationPins {
  int serialClockPin;
  int serialDataPin;
  int latchClockPin;
};

class LatchedShiftRegisterChannel {
  public:
    ~LatchedShiftRegisterChannel(); // Destructor
    LatchedShiftRegisterChannel(SerialCommunicationPins pins, int
numberOfShiftRegisterBits);
    void transmit(const byte transmissionBuffer[], int transmissionSizeInBits) const;
    void setNumberOfActiveDigitalOutputs(int numberOfActiveShiftRegisterBits);
  private:
    LatchedShiftRegisterChannel();
    LatchedShiftRegisterChannel(const LatchedShiftRegisterChannel&);
    LatchedShiftRegisterChannel& operator = (const LatchedShiftRegisterChannel&);
    void sendASingleBit(bool digitalOutputValue) const;
    void latch() const;
    void clearShiftRegister() const;
    void prepareCommunicationPins() const;
    void makeCommunicationPinsHighImpedance() const;
    void sendPaddingBits(int numberOfBitsToTransmit) const;
    void sendBitsOneByteAtATime(const byte transmissionBuffer[], int
numberOfBitsToTransmit) const;
    void sendBitsFromOneByte(byte transmissionByte, int numberOfBitsToTransmit) const;

  private:
    SerialCommunicationPins communicationPins;
    int numberOfActiveDigitalOutputs;
    int totalNumberOfDigitalOutputs;
};

#endif
```

Following the declaration of public methods, we declare the class's private methods. First, the default constructor, copy constructor, and assignment operators are declared to prevent the compiler from creating default versions of these methods. We will not implement these methods, forcing the compiler to generate an error if a programmer tries to use them. We do this because there are no possible default values for the output ports used to drive the shift registers and because we do not want several instances of the class to drive the same set of connected shift registers. We then declare a set of methods used internally to perform the low-level output operations. These methods will be explained later in this post when we describe their implementation.

Finally, the class definition contains the variables associated with instantiations of the class. First, `communicationPins` contains the three digital output ports used to drive the serial clock, data, and latch clock pins of the shift registers. `numberOfActiveDigitalOutputs` contains the actual number of digital outputs that are to be driven by the shift registers' output. `totalNumberOfDigitalOutputs` contains the number of digital outputs that could be driven if all of the shift registers' outputs were used.

LatchedShiftRegisterChannel.cpp

The `LatchedShiftRegisterChannel.cpp` file contains the implementation of the class's methods. As usual, the implementation file starts by including the header file containing the class definition. Then we have the declaration of `BITS_IN_BYTES`, a constant declaring the number of bits in a byte. We then have the declaration and implementation of a function, `clampValue()`, that takes three parameters as input: the value to be clamped, the lowest allowable value, `minimumValue`, and the maximum allowable value, `maximumValue`. The function returns either the value to be clamped, `minimumValue` if the value to be clamped is smaller, or `maximumValue` if the value to be clamped is larger. Note the presence of the `inline` keyword. It tells the compiler that this function is a good candidate to be expanded in line; instead of creating a function that is called, the compiler puts the body of the function inline, within the piece of code that calls the function. This is done to produce faster code when the function is trivial or almost trivial.

```
#include "LatchedShiftRegisterChannel.h"

const int BITS_IN_BYTE = 8;

inline int clampValue(int value, int minimumValue, int maximumValue)
{
  int clampedValue = value;
  if (clampedValue > maximumValue)
  {
    clampedValue = maximumValue;
  }
  if (clampedValue < minimumValue)
  {
    clampedValue = minimumValue;
  }
  return clampedValue
}
```

LatchedShiftRegisterChannel Constructor and Destructor

The first implemented public methods are the `LatchedShiftRegisterChannel` constructor and destructor. First, the destructor. It is called when an instance of the class is either deleted or goes out of scope. The destructor clears the shift register's output by setting all its bits to zero. It then makes the communication pins high impedance by setting the three communication pins' mode to `INPUT`. In the current program, the destructor is never called as the class instance never goes out of scope.

```
LatchedShiftRegisterChannel::~LatchedShiftRegisterChannel()
{
  clearShiftRegister();
  makeCommunicationPinsHighImpedance();
}

LatchedShiftRegisterChannel::LatchedShiftRegisterChannel(SerialCommunicationPins pins, int
numberOfShiftRegisterBits)
{
  communicationPins = pins;
  totalNumberOfDigitalOutputs = clampValue(numberOfShiftRegisterBits, 0,
MAXIMUM_SERIAL_BITS);
  numberOfActiveDigitalOutputs = totalNumberOfDigitalOutputs;
  prepareCommunicationPins();
  clearShiftRegister();
}
```

The constructor is called when an instance of the `LatchedShiftRegisterChannel` class is created. It has two arguments as input: `pins`, a structure containing the three pins to be used to communicate with the shift registers and `numberOfShiftRegisterBits`, an integer containing the number bits supported by the connected shift registers. The constructor saves the communication pins, `communicationPins`, and the total number of digital outputs, `totalNumberOfDigitalOutputs`. It limits its value to be positive and less than `MAXIMUM_SERIAL_BITS`. `numberOfActiveDigitalOutputs` is set to `totalNumberOfDigitalOutputs` as we assume that all shift register bits will be used. The constructor then prepares the communication pins to be output to and clears the shift registers.

transmit()

The `transmit()` public method is used to transmit information to the shift registers. It takes two arguments: `transmissionBuffer` and `transmissionSizeInBits`. The `transmissionBuffer` argument is an array of bytes stored least significant byte first. `transmissionSizeInBits` is the number of bits the `transmissionBuffer` byte array contains. First, the number of bits to send is clamped to the acceptable range between 0 and the number of active digital outputs. The bits are then serially sent to the shift registers, one bit at a time. If the transmission size is less than the number of active bits, padding bits are sent to ensure that all active bits have been addressed. Finally the shifted bits are latched in the shift register's output registers.

```
void LatchedShiftRegisterChannel::transmit(const byte transmissionBuffer[], int
transmissionSizeInBits) const
{
  int bitsToSend = clampValue(transmissionSizeInBits, 0, numberOfActiveDigitalOutputs);
  sendBitsOneByteAtATime(transmissionBuffer, bitsToSend);
  int paddingBitsToSend = numberOfActiveDigitalOutputs - bitsToSend;
  sendPaddingBits(paddingBitsToSend);
  latch();
}
```

setNumberOfActiveDigitalOutputs()

The number of active digital outputs can be smaller than the number of bits supported by the shift registers. The number of active digital outputs does not have to be a multiple of eight.
The `setNumberOfActiveDigitalOutputs()` public method sets the number of active digital outputs according to the `numberOfActiveShiftRegisterBits` argument. The value is clamped between 0 and the total number of digital outputs.

```
void LatchedShiftRegisterChannel::setNumberOfActiveDigitalOutputs(int
numberOfActiveShiftRegisterBits)
{
  numberOfActiveDigitalOutputs = clampValue(numberOfActiveShiftRegisterBits, 0,
totalNumberOfDigitalOutputs);
}
```

Internal LatchedShiftRegisterChannel methods

All other methods implemented are private to the class. They implement the low level work required by the public methods: the constructor and destructor methods, the `transmit()` method and the `setNumberOfActiveDigitalOutput()` methods.

The `prepareCommunicationPins()` method puts the three Arduino digital pins used to communicate with the shift registers in `OUTPUT` mode and sets their value to `LOW`.
The `makeCommunicationPinsHighImpedance()` method sets the three Arduino digital pins in `INPUT` mode. The `clearShiftRegister()` method sends `LOW` values serially to all shift register bits and latches the value in the output registers. None of these three methods take arguments nor do they return values.

The `sendBitsOneByteAtATime()` method takes two arguments: `transmissionBuffer`, a byte array containing the bits to be sent stored least significant bits in the least significant byte first (little endian), and `numberOfBitsToTransmit`, an integer containing the number of bits to send. The method first sends the bits from all complete full least significant 8-bit bytes, then the most significant bits left in the most significant byte.

```
void LatchedShiftRegisterChannel::prepareCommunicationPins() const
{
  pinMode(communicationPins.serialClockPin, OUTPUT);
  pinMode(communicationPins.serialDataPin, OUTPUT);
  pinMode(communicationPins.latchClockPin, OUTPUT);
  digitalWrite(communicationPins.serialClockPin, LOW);
  digitalWrite(communicationPins.serialDataPin, LOW);
  digitalWrite(communicationPins.latchClockPin, LOW);
}
```

```
void LatchedShiftRegisterChannel::makeCommunicationPinsHighImpedance() const
{
  pinMode(communicationPins.serialClockPin, INPUT);
  pinMode(communicationPins.serialDataPin, INPUT);
  pinMode(communicationPins.latchClockPin, INPUT);
}

void LatchedShiftRegisterChannel::clearShiftRegister() const
{
  for (int i = 0; i < totalNumberOfDigitalOutputs; i++)
  {
    sendASingleBit(LOW);
  }
  latch();
}

void LatchedShiftRegisterChannel::sendBitsOneByteAtATime(const byte transmissionBuffer[],
int numberOfBitsToTransmit) const
{
  int bytesToSend = numberOfBitsToTransmit / BITS_IN_BYTE;
  for (int byteIndex = 0; byteIndex < bytesToSend; byteIndex++)
  {
    sendBitsFromOneByte(transmissionBuffer[byteIndex], BITS_IN_BYTE);
  }
  int bitsLeft = numberOfBitsToTransmit % BITS_IN_BYTE;
  if (bitsLeft > 0)
  {
    sendBitsFromOneByte(transmissionBuffer[bytesToSend], bitsLeft);
  }
}

void LatchedShiftRegisterChannel::sendBitsFromOneByte(byte transmissionByte, int
numberOfBitsToTransmit) const
{
  byte byteToSend = transmissionByte;
  int bitsToSend = clampValue(numberOfBitsToTransmit, 0, BITS_IN_BYTE);
  for (int bitIndex = 0; bitIndex < bitsToSend; bitIndex++)
  {
    sendASingleBit((byteToSend & 1) == 1);
    byteToSend >>= 1;
  }
}

void LatchedShiftRegisterChannel::sendPaddingBits(int numberOfBitsToTransmit) const
{
  for (int i = 0; i < numberOfBitsToTransmit; i++)
  {
    sendASingleBit(LOW);
  }
}

void LatchedShiftRegisterChannel::sendASingleBit(bool digitalOutputValue) const
{
  digitalWrite(communicationPins.serialDataPin, digitalOutputValue);
  digitalWrite(communicationPins.serialClockPin, HIGH);
  digitalWrite(communicationPins.serialClockPin, LOW);
}
```

```
void LatchedShiftRegisterChannel::latch() const
{
  digitalWrite(communicationPins.latchClockPin, HIGH);
  digitalWrite(communicationPins.latchClockPin, LOW);
}
```

The `sendBitsFromOneByte()` method takes two arguments: `transmissionByte`, a byte containing bits to transmit, and `numberOfBitsToTransmit`, an integer containing the number of bits to transmit within the byte, least significant bit first. The method first clamps the number of bits to send between 0 and 8, then sends each bit, least significant bit first to the shift register. The `sendPaddingBits()` method takes one argument: `numberOfBitsToTransmit`, an integer specifying the number of padding bits to send. The method sends the specified number of `LOW` values to the shift register.

The `sendASingleBit()` method takes one argument: `digitalOutputValue`, a Boolean value specifying whether to send a `HIGH` or a `LOW`. The method sets the serial data pin to the value to output then toggles the serial clock pin `HIGH` then `LOW`. The `latch()` method takes no argument and toggles the latch clock pin `HIGH` then `LOW`.

The main Sketch

The main sketch first includes the `LatchedShiftRegisterChannel.h` file which contains the `LatchedShiftRegisterChannel` class definition. It then defines the constants used throughout the program. The serial data, serial clock and latch pins are defined as pins 2, 3, and 4 respectively. The number of LEDs is set to 10, the number of digital outputs is set to 16, the total number of outputs possible with two SN74HC595 shift register integrated circuits. The number of bytes corresponds to the number of shift register integrated circuits. The `SerialCommunicationPins` structure is initialized with the serial data, serial clock and latch pins. `WAIT_TIME`, is set to 100 milliseconds. The largest value is set to all LEDs lit, the least significant 10 bits set or $2^{10} - 1$ (`0x03FF` in hexadecimal).

The `Leds` object is instanced from the `LatchedShiftRegisterChannel` class using the serial communication pins and the number of LEDs. The `displayValue`, the value to be output to the LEDs is set to 1 and the array of bytes to be sent to the shift registers is declared.

```
#include "LatchedShiftRegisterChannel.h"

// Create bar graph display instance including shift register control pins
const int SERIAL_DATA_DIO = 2;         // Serial data input  - SN74HC595 pin 14
const int SERIAL_CLOCK_DIO = 3;        // Serial data clock  - SN74HC595 pin 11
const int LATCH_SIGNAL_DIO = 4;        // Output latch clock - SN74HC595 pin 12
const int NUMBER_OF_LEDS = 10;         // Number of LEDs to drive
const int NUMBER_OF_DIGITAL_OUTPUTS = 16; // Number of available shift register digital
outputs
const int NUMBER_OF_BYTES = NUMBER_OF_DIGITAL_OUTPUTS/8;
const SerialCommunicationPins COMMUNICATION_PINS = {SERIAL_CLOCK_DIO, SERIAL_DATA_DIO,
LATCH_SIGNAL_DIO};
const int WAIT_TIME = 100;             // 100 milliseconds per increment to get a one
second full cycle
const int LARGEST_VALUE = (1 << NUMBER_OF_LEDS) - 1;
LatchedShiftRegisterChannel Leds(COMMUNICATION_PINS, NUMBER_OF_DIGITAL_OUTPUTS);
```

```
int displayValue = 1;
byte outputByteArray[NUMBER_OF_BYTES];
```

setup()

The `setup()` method is called once before the `loop()` method is called. It sets the number of outputs of the shift register channel to the number of LEDs to drive.

```
void setup() {
  Leds.setNumberOfActiveDigitalOutputs(NUMBER_OF_LEDS);
}
```

loop()

The `loop()` method is repeatedly called indefinitely after the `setup()` method has been called. It repeatedly outputs the value to the LEDs, shifts the value left, making each LED light up in turn, then waits the specified amount of time. The delay being 100 milliseconds ends up causing all LEDs to light up every second.

```
void loop() {
  outputValueToLeds();
  shiftValueLeft();
  delay(WAIT_TIME);
}
```

Other methods in the main sketch

There are three other methods in the main sketch. The first one, `outputValueToLeds()`, copies the value to be output into a byte buffer and then transmits the value to the shift register channel.
The `moveIntegerIntoOutputByteArray()` method splits the integer specified as an argument into bytes, least significant byte first, and it copies the bytes into the byte buffer. The `shiftValueLeft()` method shifts the value to be output left by one bit. If the value is larger than the number of LEDs would allow, it is initialized to 1.

```
void outputValueToLeds()
{
  moveIntegerIntoOutputByteArray(displayValue);
  Leds.transmit(outputByteArray, NUMBER_OF_LEDS);
}
```

```
void moveIntegerIntoOutputByteArray(int integer)
{
  for (int i = 0; i < NUMBER_OF_BYTES; i++)
  {
    outputByteArray[i] = integer & 0x0ff;
    integer >>= 8;
  }
}

void shiftValueLeft()
{
  displayValue = displayValue << 1;
  if (displayValue > LARGEST_VALUE)
  {
    displayValue = 1;
  }
}
```

Putting It All Together

Build the circuit shown above and connect the Arduino Uno to your computer using a USB cable. On the computer, using the Arduino IDE, copy and paste the code above into a new sketch and files, or get a copy of the files from GitHub at https://github.com/lagacemichel/UsingShiftRegistersToIncreaseDigitalOutputs. Compile and download the sketch on the Arduino board and notice the chasing LED pattern that is displayed on the LEDs.