Arduino for Pd'ers

A tutorial on the communication between the Pure Data computer music system and the Arduino prototyping platform

written by Alexandros Drymonitis

Contents:

Introduction

This is a tutorial that tries to give some insight on the basics of the Arduino programming language to Pure Data users, by focusing on projects that are based on the communication between the two platforms. It is a result of personal exploration in an effort to realize personal projects. These projects needed to go beyond the use of Pduino and the Firmata firmware. During the course of this exploration I realized that the information I needed was scattered and usually aimed at different platforms, not Pd. For this reason I had to do a thorough search every time I wanted to write code appropriate for communication with Pd and I also had to 'translate' code that aimed at different software, or try to figure out a way to make my code executable and able to communicate with Pd.
Having reached a point where I feel rather comfortable in creating projects for Arduino and Pd, with all the above in mind, I decided to put together most of the things I've learned and achieved so far into a single tutorial, in hope that it will be helpful to others. I have to point out that all the code – both for Arduino and Pd – written in this tutorial is a result of self tuition. Therefore some terms or even concepts might have been wrongly interpreted and/or expressed. Still, the code you'll find here is fully functional. If you find a mistake, or if you want to contact me for any reason, send me an email at alexandros[at]drymonitis.me.

In this tutorial we'll learn how to write Arduino sketches and Pure Data patches that work together. The tutorial will be split into several small projects, starting from very basic use (e.g. one digital pin) and adding complexity as we go. We will not use the Firmata firmware, but instead we will write our own firmware which will be narrowed down to specific tasks, according to the needs of the respective project.

In order to use the code provided, you'll have to download and install the Arduino IDE from Arduino's website, as well as Pd-extended. Also, some experience in Pd programming is assumed.

Note – the font for Arduino sketches in this tutorial will be the following

```
int variable;
```

As for Pd patches, they will either be illustrated as an image, or with ASCII as follows:

[comport] is the comport object; any object in ASCII will be between square brackets

[open 6(is the message "open 6"; any message in ASCII will be between a square bracket and an opening parenthesis

[0 \ is a number atom

[x] is a toggle

| is a connection wire. For example the message and object above will be connected like this:

```
[open 6(
|
[comport]
```

All circuit image are made with Fritzing
This tutorial was written in LibreOffice

Project 1 – digital output

In our first project we will replace Arduino's blink example with a combination of Arduino code and a Pd patch. We will set Arduino's pin 13 (which includes an integrated LED) as output in the Arduino code and we will control its state from the respective Pd patch.

Lets write our code for the Arduino. When programming the Arduino you need at least two functions in your code. A function is distinguished by the parenthesis that follows the function's name. For example:

```
void setup()
```

is the setup function that is obligatory to all Arduino programs. You might have noticed that the parenthesis doesn't contain anything. This is because it doesn't take any arguments. We will leave the discussion as to what `void` means for later.
The two standard and obligatory functions in Arduino programming are `void setup()` and `void loop()`. The setup function will run only once, when the Arduino boots. This is the function where we'll initialize various stuff of our program. The loop function will start running immediately after the setup function and, as its name indicates, it will loop and run for as long as the Arduino is powered. A great number of Arduino sketches use only these two functions, so for starters they are more than enough.

Our blinkPd code will be the following:

```
// set a variable to hold the LED's pin number
int led = 13;

void setup()
{
  // set pin 13 as output, to light up the LED
  // whenever it is told so from Pd
  pinMode(led, OUTPUT);
  // start the serial communication so the Arduino
  // and Pd can communicate with each other
  Serial.begin(9600);
} // end of the setup function

void loop()
{
  while(Serial.available()){
    // read and store the data that comes through the serial port
    byte ledState = Serial.read() - '0';
    // set the state of the LED according to the incoming data
    digitalWrite(led, ledState);
  }
} // end of the loop function
```

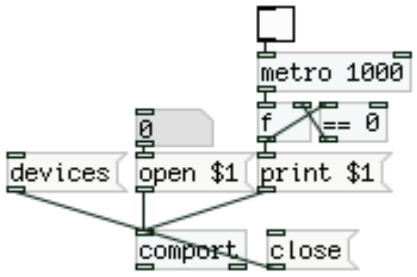
Now we'll analyze the code. This simple program contains most of the necessary functions and variables an Arduino sketch will use, so we'll analyze it in depth to gain knowledge for the projects that are yet to come.
First of all, you can see that certain words are highlighted with different colors. The double forward slash and whatever comes immediately after, will be grey. This is a

comment, much like the comments in a Pd patch. There're there to give us some insight as to what the preceding or following line of code does. The compiler will ignore anything written after // so you can literary write whatever you want.
In the second line of our code we write `int led = 13;` This way we create a variable of type int, which is an integer, called 'led' and we assign it the value 13. We will later use this variable to set pin number 13 as output (in the `setup` function) and to set the state of this pin (`HIGH` or `LOW`, according to what data Pd sends, to turn the LED on or off, in the `loop` function). When setting a variable, you always have to define its data type, be it `int`, `float` or something else. We'll go through different data types as we build more projects. Also notice the semicolon at the end of the line. The semicolon tells the compiler that this line has ended and needs to be executed.
In line 4 we declare our `setup` function. This is the way you will always call this function as it never takes any arguments.
In line 5 we have a curly bracket. Whatever is included in a function must be within curly brackets, otherwise the compiler will complain and the program won't be compiled. In line 8 we call the `pinMode` function. Notice that we have parenthesis after the keyword `pinMode` and this time the parenthesis do have arguments. Specifically, `pinMode` takes two arguments, the pin number and the mode of that pin. In our case we set the pin number 13 (using the variable `led`) as `OUTPUT`, written in capital letters, always closing our line with a semicolon. In this specific simple example we could have avoided declaring a variable for the pin number, but this way the code is more intuitive, plus we're getting the hang of using variables.
In line 11 we call the function to start the serial communication, so that Pd and Arduino can talk to each other. This function takes one argument which is the baud rate. The baud rate is the rate of data per second transferred in the serial line. In this case we exchange data between Arduino and Pd 9,600 times per second.
In line 12 we have the closing curly bracket, and this is where our setup function ends.
In line 14 we call our `loop` function, which will follow immediately after the setup function is finished. In line 15 we open the curly bracket that will include whatever this function is supposed to do.
Line 16 is calling the `while` control structure, which is a loop (yup, it's a loop within a loop). What the `while` loop does, is test the statement within its parenthesis, if the statement is true, then the code within the loop's curly brackets will be executed. Afterwards, `while` will test the statement again and act accordingly. This means that the `while` loop will keep on executing its code for as long as its statement is true. More specifically, in our code the `while` loop is declared like this `while(Serial.available())`. We see that the statement of the loop is a function itself, `Serial.available()` with no arguments. This function is true for as long as there is data coming in from the serial port. So as soon as we send something from Pd, this function will be true and the `while` loop will execute the code within its curly brackets.
In line 18 we declare a variable of the type `byte`, named `ledState` and assign it the value read from the serial port. We achieve this by calling the `Serial.read()` function, which returns bytes read from the serial port. But there is something tricky there. The line is written like this `byte ledState = Serial.read() - '0';` `'0'` is the ASCII value of the number 0, which is 48[1]. Since any incoming data will have their ASCII value, if we send a zero, Arduino will collect the number 48, and if we send a one, it will collect the number 49. Subtracting ASCII zero, which is 48, we will collect either 0 or 1.
In line 20 we call the `digitalWrite` function which sets the state to a digital pin. This

1 Try it in a Pd patch. Connect [key] to a number atom, press 0 on your keyboard and check the output.

state can be either HIGH or LOW, expressed numerically with 1 or 0. The arguments this function takes are the pin number, set by using our led variable, and the state of that pin. Since we'll be controlling the LED from Pd, the state will be set by the data that comes in from the serial port. This data has been read and stored in line 16 as a byte in the variable ledState. So, since we have variables for both arguments of the digitalWrite function, we can safely write digitalWrite(led, ledState); never forgetting the semicolon. And this concludes our Arduino sketch. Take a moment to count the curly brackets closing the program. They're two, one for the while loop and one for the main loop function, void loop(). As already mentioned, failing to close an open curly bracket will result in a compilation error and your program will not compile.
One last thing before we move on is to explain the void data type. This is a type used in functions that don't return anything. Both functions, setup and loop, are of this type. There are other functions though, built in the Arduino language that do return something. For example, Serial.read() returns the first byte read from the serial port. This function is of the type int, and returns a byte (no idea why it's int and it returns a byte, but that's how it is). So its declaration in Arduino's source code is more or less like this: int Serial.read(); (I say more or less, because it's more complicated than this, but we can still get an impression of how things work). Hence line 18 assigns the value returned from Serial.read() to ledState. But setup and loop return nothing so they are of the type void.

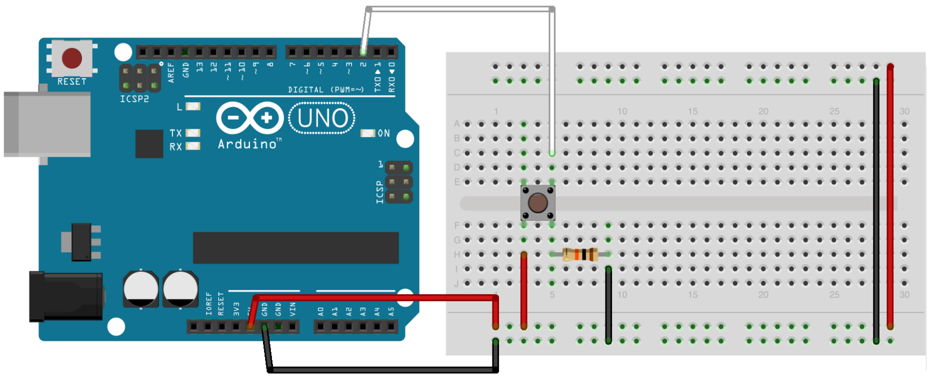
We have now written our first sketch using various functions built in the Arduino language. In later projects we'll go through the use of functions we will define ourselves. Now lets see the corresponding Pd patch. It is quite simpler than the code we already wrote, plus it won't need too much commenting as we already know how to patch. The patch looks like this:



Since we'll be using a baud rate of 9600, we don't need to specify it as an argument to [comport], as it's the default one. If you send [devices(to [comport] you'll see the available serial devices in Pd's console. Arduino's serial line should be something like /dev/tty.usbmodemxxx on OS X, /dev/ttyACMx on Linux and COMx on Windows, where x is a number. Each serial device will be assigned a number. Write Arduino's number to the number atom. This way you'll open the communication between Pd and that serial line (sending "close" will close it). If you click on the toggle, you'll start sending the message "print 1" and "print 0" to the serial port sequentially every second, which is what the Blink sketch, which is included in Arduino's examples, does. By prepending the word print to the data you send, you translate your data to its ASCII value, similar to the Serial.print function of the Arduino language. You can also send a simple 1 or 0, and avoid subtracting '0' in the Arduino code (Serial.read(); alone instead of Serial.read() - '0'; in line 18). Using print though is better practice as it is a much more useful method, if not the only method, when things become more complex.
If you check the integrated LED on your Arduino board, you'll see it blinking every one second. You can instead use an external LED by plugging its long leg to pin 13 and its short to the pin right next to it, writing GND (GND stands for ground).

Project 2 – digital input

In our next project we'll be sending digital values received from a push button in Arduino to Pd. For this project we'll have to do some wiring to make a circuit that works. We'll use a push button to give input to the Arduino and send that to Pd. You can see the circuit's image below.



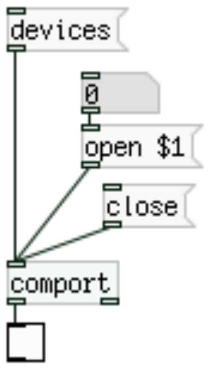
This circuit is said to have a pull-down resistor, because the 10kOhm resistor connects the push button and Arduino's digital pin to ground. If it were connecting them to 5V it would be a pull-up resistor. It is said that pull-up resistors provide better stability in a circuit, but the readings of a switch are reversed, meaning that when you press the button the controller's pin will read LOW and vice versa. For this project we'll use a pull-down resistor, but for future projects we'll use pull-up resistors.

Lets write the code.

```
1.  // set a variable to hold the button's pin number
2.  int pushButton = 2;
3.
4.  void setup()
5.  {
6.    pinMode(pushButton, INPUT);
7.    Serial.begin(9600);
8.  }
9.
10. void loop()
11. {
12.   // read and store the data from the push button
13.   byte buttonState = digitalRead(pushButton);
14.   // write the stored data to the serial port
15.   Serial.write(buttonState);
16. }
```

The code's size is more or less the same with our previous project's code and quite similar too. We reduced the comments since many functions have already been explained. The first difference is in line 6, where we call the `pinMode` function but this time we set the pin as `INPUT`, since we'll be receiving input from the push button.
In line 13 we declare a variable of type `byte`, called `buttonState`, and we assign it to the value read from the digital pin 2. We achieve that by calling the `digitalRead` function. This function is quite similar to `digitalWrite`, but it takes only one argument, the pin to read – in this case the `pushButton` variable.
In line 15 we call the `Serial.write` function, which, like `Serial.read`, is part of the

Serial class[2]. This function writes a byte to the serial port. It is different from the Serial.print function as the latter writes the data to the serial port with their ASCII values. So, if we used that one instead, in Pd we would be receiving a 49 and a 48 according to the state of the push button, instead of a 1 or 0.
And this is the end of our code. Arduino will be sending the data received from the digital pin 9,600 times a second to Pd. Now lets look at our Pd patch.



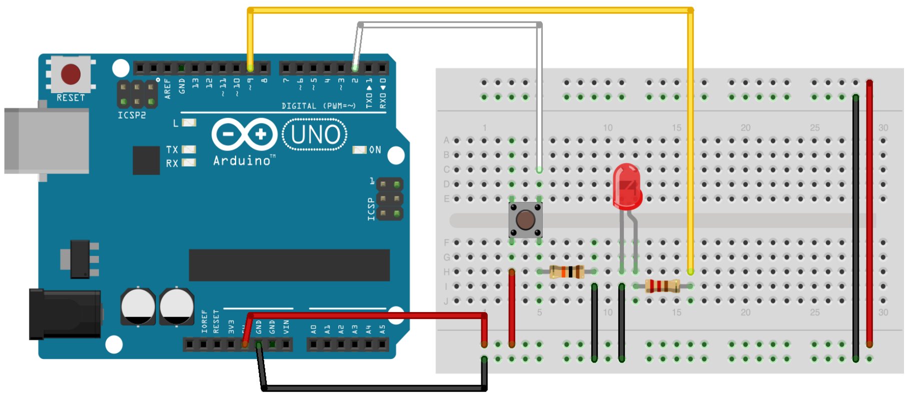
Our patch is even simpler than our first project's patch. Again we can check Pd's console to see which port number Arduino is at, by sending [devices(to [comport]. We can then open that serial port by typing the port's number to the number atom. If we press the button on the circuit board, we should see the toggle going on and off accordingly.

We have now completed our second project, but before we move on, lets combine the first two projects by adding an LED to the circuit which will be controlled by the push button. We will add a few things to our code so that Pd won't interfere to control the LED as we can control it immediately within the Arduino sketch. So the code will look like this:

```
// set a variable to hold the button's pin number
int pushButton = 2;
// set a variable to hold the LED's pin number
int led = 9;

void setup()
{
  pinMode(pushButton, INPUT);
  pinMode(led, OUTPUT);
  Serial.begin(9600);
}

void loop()
{
  // read and store the data from the push button
  byte buttonState = digitalRead(pushButton);
  // write the button state to the LED
  digitalWrite(led, buttonState);
  // write the stored data to the serial port
  Serial.write(buttonState);
}
```

2 The Serial class contains functions like, begin, write, read etc. Check Arduino's reference for more information

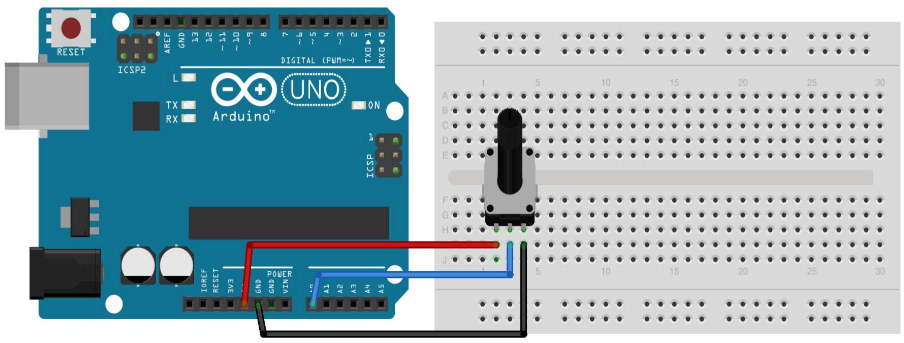
And this is our code. We have added only a few lines of code. In line 4 we set a variable to hold the LED's pin number. We didn't use 13 this time to do some more wiring in our circuit.
In line 9 we call the `pinMode` function to set the LED's pin as `OUTPUT`, and in line 18 we call the `digitalWrite` function to set the LED's state. Since we have already read and stored the push button's state in the `buttonState` variable, all we need to do is write `led` and `buttonState` as arguments for the function.
Below you can see the circuit.



We need to explain the LED circuitry since it's a bit different from our first project's circuit. We have placed the LED on the breadboard and since we're not using the integrated resistor on pin 13[3], we need to place a resistor on the circuit. So we need to connect the LED's short leg to ground and its long leg to a 220 Ohm resistor. The other end of the resistor will go to the pin that will control the LED, in this case pin 9, and that's it. You can now test the code and circuit along with the Pd patch.

Project 3 – analog input

In this project we'll use a potentiometer and receive its values in Pd. A potentiometer is actually a variable resistor. Depending on its position, it applies some resistance to the current going through it. Digitizing its signal we receive a certain value for the resistance the potentiometer is currently applying. The circuit is quite simple, you can see it in the image below.



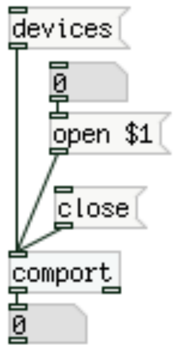
What happens here is that we supply the legs at the sides with 5V and ground, and we connect the middle leg with Arduino's analog pin. This potentiometer is 10kOhm.

Lets see the code which is a bit tricky. Our first attempt is the following:

3 All digital pins have 20-50kOhm integrated resistors (pin 13 has a smaller one), but this is too much resistance to light up an LED, plus it's better to really build the circuit physically to get the hang of it

```
// variable to hold the knob's pin number, analog pin 0
int knob = 0;

void setup()
{
  Serial.begin(9600);
}
// we don't need to call pinMode as the analog pins are
// input pins only

void loop()
{
  // call the analogRead function to read the value
  // of a specific pin
  byte knobValue = analogRead(knob);
  Serial.write(knobValue);
}
```

We can try to use this code along with the following Pd patch.



But if we spin the potentiometer we see something strange happening. The values received go from zero to 255 and then wrap back to zero. This happens four times and it's because the Serial.write function is transferring bytes to the serial line, as we mentioned in the last project. But Arduino's analog pins have a 10-bit resolution, whereas a byte consist of eight bits only. So, sending a 10-bit value through a serial line, will wrap this value to a byte, from zero to 255 instead of 1023 which is ten bits. There are a couple of ways to overcome this problem. One is using the Serial.println function, which sends all data in their ASCII value followed by a new line character, which is ASCII 10. The problem with this technique is that we'll have to assemble the ASCII values received in Pd to their decimal counterparts and this is rather cumbersome. An easier way is to disassemble the 10-bit value and transfer two bytes over to Pd which we will re-assemble to retrieve the 10-bit resolution.
Even though this technique is easier than using the Serial.println function, there's still a bit of work to do. First of all, since we'll be sending more than one value, we have to introduce the array in the Arduino language. The array is much like the array in Pd, only there's no graphing of its elements in the Arduino code. An array is a collection of numbers which we can access by means of indexing. So, the first value in an array has the index 0, the second has the index 1, etc. In Arduino we declare arrays like this:

```
byte myArray[2];
```

This way we declare an array with two elements of the data type byte[4] (remember, Serial.write writes bytes to the serial line), without assigning any values to its elements yet. What we need to do with the value received from the analog pin is mask it to a range within a byte, preferably less than 255 (we'll explain why in a bit), and then use a bitwise operation to increment a counter every time our value wraps back to zero. We can do this like this:

```
myArray[0] = analogRead(0) & 0x007f;
myArray[1] = analogRead(0) >> 7;
```

The first line masks the value of the analog pin to a 7-bit range, from zero to 127. 0x007f is the hexadecimal version of 127 declared as an int (an int has the size of two bytes; in hexadecimal, two digits cover the range of one byte). The second line shifts the bits of the value of the analog pin by seven positions to the right[5], which results in an incrementing counter whenever the first element of the array wraps back to zero. So our Arduino code could be modified as follows:

```
// variable to hold the knob's pin number, analog pin 0
int knob = 0;
// declare the array to be transferred over serial
byte myArray[2];

void setup()
{
  Serial.begin(9600);
}
// we don't need to call pinMode as the analog pins are
// input pins only

void loop()
{
  // assign values to the array's elements
  myArray[0] = analogRead(knob) & 0x007f;
  myArray[1] = analogRead(knob) >> 7;
  Serial.write(myArray, 2);
}
```

Notice the pre-last line of our code, where we call the Serial.write function. This time we give two arguments, one is the array to be transferred and the other is the size of the array. This is necessary when writing arrays to the serial line.
There are two problems with this code though. First of all we call the analogRead function twice, which makes our code slower. Second, since the two values sent to Pd do different things (we'll see what each does when we cover the corresponding Pd patch), it's very likely to encounter a problem as there's nothing to denote the beginning of the data stream, which can lead to reversing the data, receiving the 2nd value of the array first and the 1st value afterwards.
To solve these two problems we need to modify our code as follows:

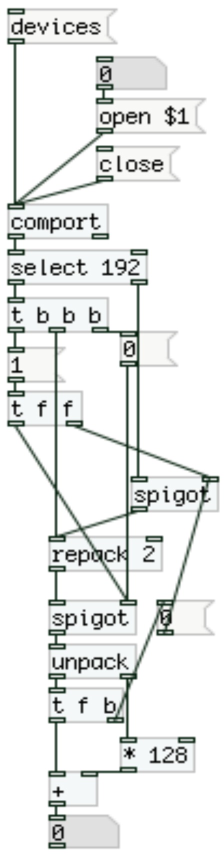
```
// variable to hold the knob's pin number, analog pin 0
int knob = 0;
// declare the array to be transferred over serial
```

4 If you're having difficulties grabbing what an array is, think of it as a list, for now
5 Check Arduino's reference to see what bitwise AND (&) and bitshift right (>>) do

```
4.  byte myArray[3];
5.
6.  void setup()
7.  {
8.    Serial.begin(9600);
9.  }
10.
11. void loop()
12. {
13.   // set the first byte of the array to the hexadecimal value 192
14.   myArray[0] = 0xc0;
15.   int knobValue = analogRead(knob);
16.   myArray[1] = knobValue & 0x007f;
17.   myArray[2] = knobValue >> 7;
18.   Serial.write(myArray, 3);
19. }
```

In line 14 we assign the value 192 (in hexadecimal) to the first element of the array, which will denote the beginning of the data stream in the Pd patch. This value has to be unique and since Serial.write writes bytes, which take values from zero to 255, we need to make sure that no other value of the array will reach this high. This is why we're masking the pin's value to a 7-bit range (zero to 127) and not to an 8-bit, as if we did the latter, it would be very likely that the value of the array's second element could, at some point, be the same as the value that denotes the beginning of the data stream, thus confusing Pd and distorting the data[6].

The Pd patch is a bit more complicated than the ones we've already made.

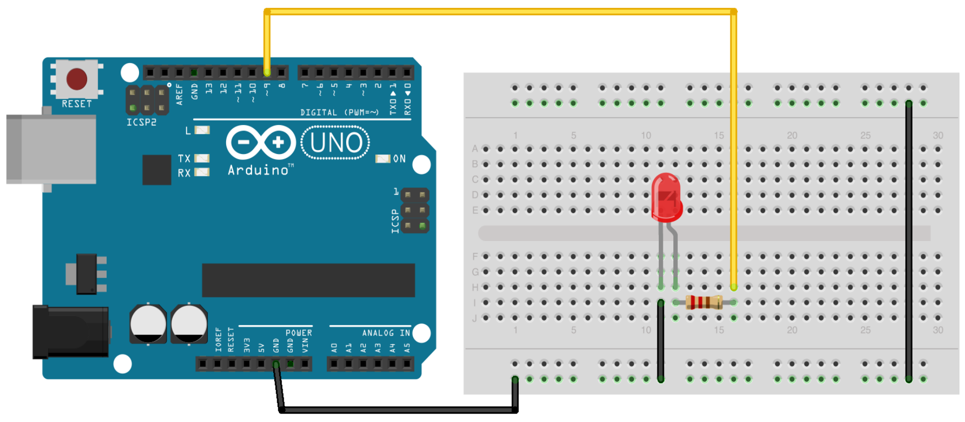


6 I have copied and slightly modified this technique from Open Music Lab's rePatcher

Below [comport] we can see [select 192], which will give a bang out its left outlet when it receives the value 192, which is the value that denotes the beginning of the data stream. This bang goes to four destinations, it first closes the lower [spigot], it bangs [repack 2][7] to clear it (the [spigot] below is closed, so [repack]'s data won't go anywhere, it will just get cleared), and finally it opens both [spigot]s. The rest of the data received from [comport] go out [select]'s right outlet to the upper [spigot] which lets all numbers go through to [repack] as [spigot] is now open. [repack]'s argument is very important, it has to be one less than the number of data sent from Arduino, as the first value is absorbed by [select 192]. In our case it's 2. Once [repack] receives two values, it outputs them as a list, and since the lower [spigot] is now open, the list will go to [unpack]. [unpack] will send the second number of the list to [* 128] and the first to [t f b], which first closes the upper [spigot] and then outputs the float it received to [+]. Remember that the first value is the analog pin's value masked to a 7-bit range, going from zero to 127 repeatedly, as we spin the potentiometer. The second value, which goes to [* 128] is the counter that increments by one whenever the first value wraps back to zero. So in the beginning, the second value is 0 * 128 = 0, which is added to the first value that goes from 0 to 127. When the first value will wrap back to zero, the second value will be 1 * 128 = 128. Adding this to 0 gives us 128, which is the next value of 127. The second time the first value will wrap back to zero, the second value will be 2 * 128 = 256, which is the next value of 255 (127 + 128), and so on, till we get 1023. This way we can retrieve the 10-bit resolution of Arduino's analog pins from it's 8-bit serial communication. Of course we can use any kind of sensor instead of a potentiometer.

Project 4 – analog output

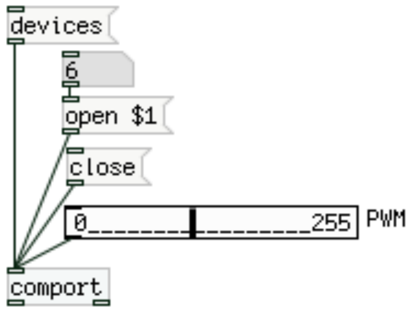
In this project we'll use Arduino's analog output, which isn't analog really, but digital that creates the illusion of analog. It's the PWM (Pulse Width Modulation) capability of some digital pins (3, 5, 6, 9, 10, 11 for Arduino UNO). PWM is essentially the same thing as the duty cycle in a square wave oscillator. Within one period of the oscillator's waveform, we determine how long the pin will be `HIGH` and how long it will be `LOW`. Lighting up an LED this way creates the illusion of dimming. If the pin that lights up the LED is 100% of its period `HIGH`, the LED is fully turned on, if the pin is 50% of its period `HIGH` and 50% `LOW`, we perceive this as the LED being half way on. What actually happens though is that the LED goes on and off repeatedly so fast that we can't actually see the difference and we perceive a situation in the middle, the LED being half way on.
This is the circuit and code:



7 This is an object from the zexy library, check its help patch to see what it does

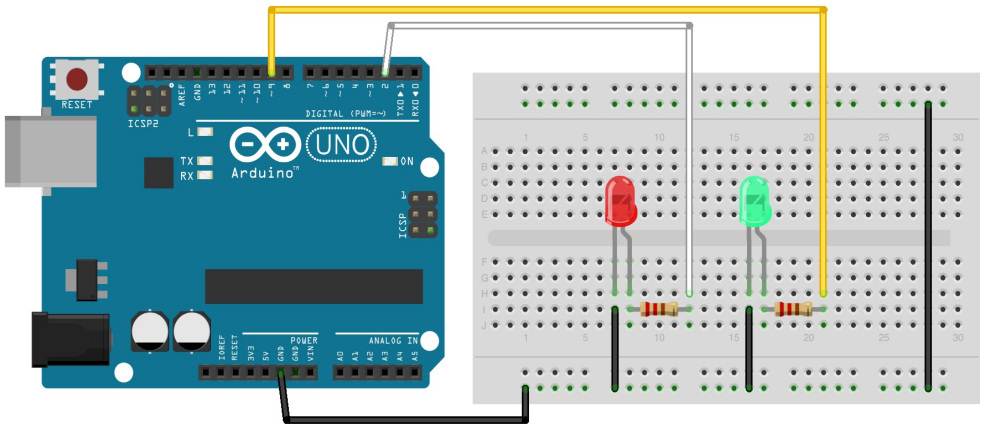
```
1.  // variable to hold LED's pin number
2.  int led = 9;
3.
4.  void setup()
5.  {
6.    pinMode(led, OUTPUT);
7.    Serial.begin(9600);
8.  }
9.
10. void loop()
11. {
12.   while(Serial.available()){
13.     byte LEDpwm = Serial.read();
14.     // use the analogWrite function for PWM
15.     analogWrite(led, LEDpwm);
16.   }
17. }
```

Our variable declaration and our `setup` function should be known by now. In the `loop` function we run the `while` loop, like we did in our first project. Inside this loop we create a variable of type `byte`, called `LEDpwm` and assign it the value read from the serial port.
In line 15 we call a new function, `analogWrite`. This function works much like its digital counterpart, `digitalWrite`. It takes two arguments, the pin to write a value to, and the value to write to that pin. In our case its the pin number held in the `led` variable, and the value held in the `LEDpwm` variable. This function though takes a byte as a value to write to the PWM pin, meaning a value between zero and 255, 255 being 100% of the function's period `HIGH`.
The corresponding Pd patch is the following:



Since we're sending a single value to the serial line from Pd, we can omit prepending the word “print” to that value before we send it to [comport]. If we use a slider (or another GUI) for our purpose, we need to change its preferences and set its range from zero to 255, in order to use Arduino's PWM full range.

Project 5 – analog and digital output
In this project we'll see how we can send two distinct values from Pd to Arduino in order to control two different tasks. We'll be controlling a PWM and a non-PWM digital pin at the same time. The non-PWM pin will be lighting up an LED when the DSP is turned on in Pd and the PWM pin will be controlling an LED according to the values of an oscillator. The code is a bit more complex than what we've already written, so we'll need to do some in depth analysis. Here's the circuit and code:



```
// variables to hold pin numbers
int pwmLED = 9;
int dspLED = 2;
// variables to hold pin states
int pwmLEDvalue;
int dspLEDstate;
// variable to hold and assemble incoming data
int temporary;

void setup()
{
  pinMode(pwmLED, OUTPUT);
  pinMode(dspLED, OUTPUT);
  Serial.begin(9600);
}

void loop()
{
   while(Serial.available()){
     byte inByte = Serial.read();
     if((inByte >= '0') && (inByte <= '9'))
       temporary = 10 * temporary + inByte - '0';
     else{
       if(inByte == 'p'){
         pwmLEDvalue = temporary;
         temporary = 0;
       }
       else if(inByte == 'd'){
         dspLEDstate = temporary;
         temporary = 0;
       }
     }
     analogWrite(pwmLED, pwmLEDvalue);
     digitalWrite(dspLED, dspLEDstate);
   }
}
```

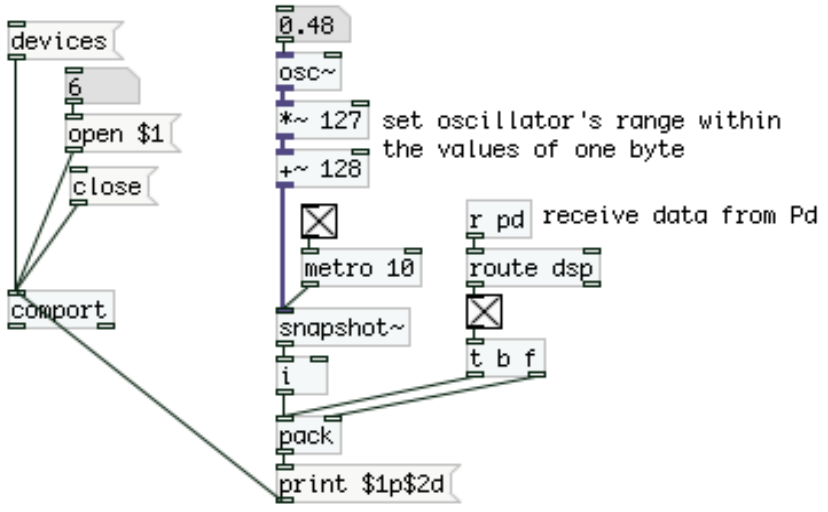
In the beginning we set some variables for various data. The comments of the code give some insight as to what each variable does. The `setup` function is nothing new. The `loop` function though has quite some new techniques.
We begin with the `while` loop, exactly the same way we've been using it so far. This time we'll be sending data from Pd in the form [print $1p$2d(, where 'p' indicates the PWM value and d the value of the LED which will turn on or off according to the DSP state in Pd.
In line 20 we assign the data that comes in the serial line to the `inByte` variable. In line 21 we use the `if` control structure. This structure is a very basic and useful one when programming. What it does is check its statement (the statement inside the parenthesis), if the statement is true, the code immediately after it, or inside the curly brackets[8], will be executed. If it's not true, the program will move on. It's very usual that an `if` statement is followed by an `else` or `else if` statement. The names of these structures are very intuitive. `else if` will test its statement if the `if` statement before it is not true, and only if `else if` is true, will its code be executed. `else` is the default condition to be executed if none of the above is true. `else if` and `else` are optional.
Lets take a hypothetical message [print 100p1d(sent from Pd to Arduino. Arduino will receive the list '49 48 48 112 49 100' which is the ASCII values of everything but "print". The first `if` statement tests whether the data is between `'0'` and `'9'`, ASCII 48 and 57, with a logical AND (&&)[9]. If this is true then the line below will be executed. This line takes the variable `temporary`, which is not initialized to any value, multiplies it by ten and adds the incoming number. Since we'll be sending data in their ASCII values, each number will come in the serial line separately. So line 21 is true for the first three numbers and line 22 will execute as follows: The first time `temporary` is 0 so we get 10 * 0 + 49 – 48 = 1. Since the next two pieces of data are numbers, this line will execute another two times, but `temporary` will now hold 1. The second time, the serial line receives 48 (the ASCII vale of 0), so line 22 will execute as follows: 10 * 1 + 48 – 48 = 10. And the third time, the serial line will receive 48 again and the line of code will execute as follows: 10 * 10 + 48 – 48 = 100. This way we'll store the number 100 to `temporary`.
The next piece of data in the serial line is `'p'`, so the `if` statement in line 21 will not hold true and the program will move on to the `else` statement. Inside the curly brackets of `else` we find an `if` and an `else if` statement. It's very common to have nested `if` statements or loops (`while`, `for` etc.). What happens when we have nested statements is that if the very first statement is true, then the code will move on to the statements inside the true statement, executing as it would for a single statement. The philosophy is pretty straight forward and intuitive.
The `if` statement in line 24 checks whether the incoming data has the ASCII value of the letter p. If this is true, then it assigns the value stored in `temporary` to the `pwmLEDvalue` variable and resets `temporary` to 0. In order for this to work, `temporary` must be a global variable, meaning it must be declared outside all scopes (functions).
The next element of our list is 1, ASCII 49 for the LED on pin 2. Our program will jump up to line 21 again to see if this is a number. Since it is, the statement in this line is true and the code below will be executed which will give 10 * 0 + 49 – 48 = 1. Remember that we've reset `temporary` to 0, so we start assembling numbers from the beginning.
The last element in the list is `'d'`, ASCII 100, so the `if` statement in line 21 is not

8 If the code to be executed is one line only, the curly brackets are not necessary. The code is indented though, for easier reading

9 Logical AND (&&), known as Boolean AND, connects two statements. Its output will be true only if both statements are true – check Arduino's reference for more information

true and the program moves on to the else statement. Inside else we check whether this data is 'p', which again is not true so we move on to else if and check whether it's 'd', which is true, so the code inside else if's curly brackets will be executed and assign temporary to the dspLEDstate variable and reset temporary to 0. After we've read all incoming data, before the while loop ends, we call the analogWrite and digitalWrite functions and set the stored values to the corresponding pins.
We see now that when we want to send various types of data to Arduino from Pd, "print" is a very effective method that helps separate values and diffuse them to any location we want.

The Pd patch for this project is quite simple, concerning the data sent to [comport]. You can see it below:

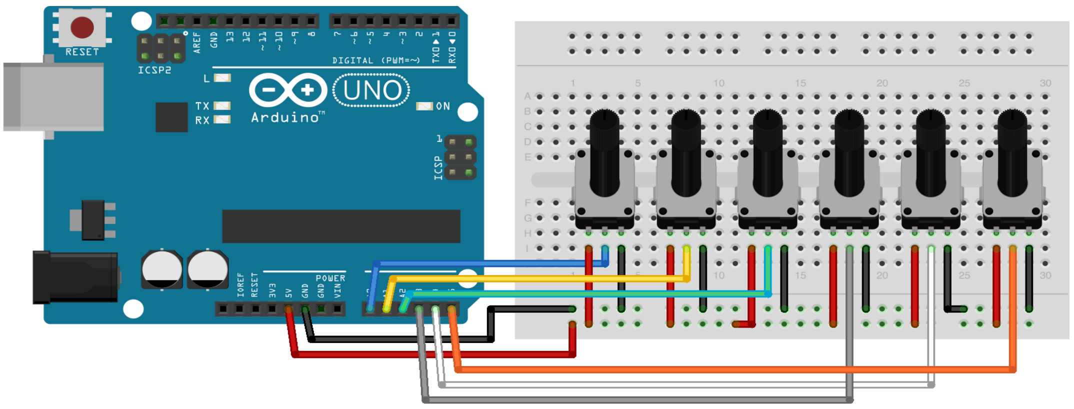


The heart, so to say, of the patch is [print $1p$2d(. We make a list of two numbers with [pack], the first number is the amplitude of the oscillator, mapped to the range of Arduino's PWM. We send this signal to [snapshot~] which we bang every ten milliseconds and send its output to [i]. This step is very important because [osc~] outputs floats and we need to send integers to Arduino. If we omit this step the result won't be the one we expect.
On the right hand side of the patch we have [r pd] which receives various messages from Pd. We connect this to [route dsp] so we can use only the information we get concerning the DSP state (whether it's on or off). We don't really need the toggle underneath, but it's there to visualize things. If we turn on the DSP in Pd, [route dsp] will output a 1, and if we turn it off it will output a 0.
If the DSP is on, then the oscillator will start outputting values and [pack] will be sending lists to [print $1p$2d(which we can see in our circuit – the LED on pin 2 will be on and the LED on pin 9 will be dimming in and out (assuming [metro] is on).
If we turn off [metro], the oscillator's LED will be still, and the DSP LED will be on or off according to the DSP state in Pd. We have connected [route dsp] to [t b f] which stores its number to [pack]'s right inlet and sends a bang to its left inlet in order to output its list, so it's independent from [osc~].

Before we move on it would be good to talk a bit about the locality of variables in Arduino. A variable is considered local within a function when it has been declared inside that function. For example, if we declare a variable at the beginning of the loop function, this variable will be valid for the whole function and functions called inside it (while, if etc.). If a variable is declared inside a while loop (which is called inside the loop function), it won't be valid once we exit the while loop and we can't call it any more. In order to be able to use a variable inside any function, we can

declare it as global. We can do that by declaring it at the beginning of our program, outside all functions. This way we get the advantage of being able to use that variable in any desired place in our code, but global variables are said to have slower access, while local variables are much easier and faster to access.
Another difference between the two is that global variables will hold their value until they're assigned a new one, while local variables will be initialized anew whenever the function in which they're declared begins, losing their previous value assignments. Lastly, a global variable will always occupy memory, while a local one will free its memory when its function has finished. Eventually it's a matter of functionality as to what kind of variables we use.

Project 6 – more than one analog inputs
In this project we'll use all six analog pins of the Arduino. For this reason we'll introduce two new methods that helps us achieve this without needing to write many lines of code. The circuit is similar to the third project's circuit, multiplied six times.



Each potentiometer has its left leg connected to 5V, its right leg to ground and its middled leg (called the wiper) to an analog pin of the Arduino. We could write code similar to the code of the third project, something like this:

```
myArray[0] = 0xc0;
int knob = analogRead(0);
myArray[1] = knob & 0x007f;
myArray[2] = knob >> 7;
knob = analogRead(1);
myArray[3] = knob & 0x007f;
myArray[4] = knob >> 7;
```

and so on. Notice that we declare the variable knob once and assign it to the value of the first analog pin, and the second time we call it, we don't need to declare its data type (int in this case), but only assign it to a new value. This code though is very inefficient as we have to repeat similar lines of code many times. Plus, it's not flexible at all, since if we want to use less pins, we'll have to erase many lines and do quite some editing in general. For this reason we'll use the for loop (another control structure). The syntax of the for loop is the following:

```
for(int i = 0; i < someNumber; i++){
  // put your code here
}
```

What this loop does is first declare a variable, in this case i, and assign it to an initial value, in this case zero. Then it moves on to the next statement – statements are separated by a semicolon – which checks if i is less than some number. If this statement is true, then the code inside the curly brackets will be executed[10]. After the code in the curly brackets has been executed, the loop moves to the last statement of the parenthesis, i++. This is a special symbol in Arduino's language (it's actually a C/C++ symbol) which increments a value by one. We could have written i = i + 1 instead, but since incrementing by one is way too common, these programming languages have this special symbol.
After i has been incremented by one (it will now hold the value of 1), the loop goes back to the test statement again and checks if the variable is still less than someNumber. If it's still true, the code inside the curly brackets will be executed again, and afterwards i will again be incremented by one. And this will go on until i is not less than someNumber. When the test statement is not true any more, the loop will end. So, our code can take a form like this:

```
myArray[0] = 0xc0;
for(int i = 0; i < 6; i++){
  int knob = analogRead(i);
  myArray[i * 2 + 1] = knob & 0x007f;
  myArray[i * 2 + 2] = konb >> 7;
}
```

Take a minute to think what exactly happens inside the loop. In order to access the right element of myArray, we need to do some simple arithmetic. When i is 0, we'll multiply it by 2 and add one, so 0 * 2 + 1 = 1, for the value that is masked to a range of 0 – 127, and then 0 * 2 + 2 = 2, for the value that increments by one whenever the first value wraps back to zero. When i is 1, we'll get 1 * 2 + 1 = 3 and 1 * 2 + 2 = 4 etc. This way we can assign the values of the analog pins to the correct elements of myArray. Still it is not very elegant plus it is a bit cumbersome. There is a method in the C programming language (and its derivatives) that can simplify this situation a lot. It's called post increment and it looks like this:

```
int index = 1;
myArray[index++] = konb & 0x007f;
myArray[index++] = knob >> 7;
```

We first declare a variable called index and assign it the value of 1. Then we use this variable to access an element of myArray, appending the double addition sign. What happens is that the variable will hold the value it has been assigned to till the line of code has been executed and then it will be incremented by one. So the line myArray[index++] = knob & 0x007f; will assign the value "knob & 0x007f" to myArray[1] and then will increment index by 1. The line below will assign "knob >> 7" to myArray[2] and then will increment index by 1 again. In the code below we can see that this tool is very effective, leaving little room for errors when we write code like this. Applying it to our project our code can take the following form:

```
myArray[0] = 0xc0;
int index = 1;
for(int i = 0; i < 6; i++){
  int knob = analogRead(i);
```

10 Again, if the code to be executed is one line only, the curly brackets are not necessary

```
    myArray[index++] = knob & 0x007f;
    myArray[index++] = knob >> 7;
  }
```

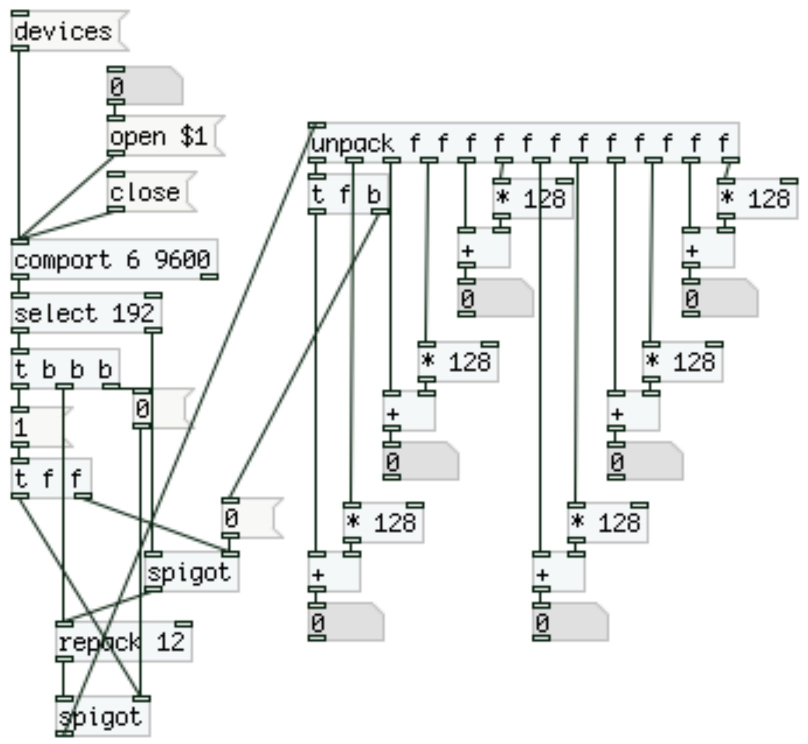
Note that we declare and initialize `index` outside the for loop, as if we did that inside the loop, `index` would be initialized to 1 at each iteration and the post increment technique wouldn't work. Also note that `i` is valid only inside the scope of the `for` loop and doesn't exist outside of it. Still, we can use it any way we want while we're in the loop.

This code is much more efficient, elegant and readable. It will store 13 values to `myArray` without any conflict. Take a minute to analyze it and understand what it does exactly. The full code for this project is this:

```
1.  // declare the array to hold the analog pins values
2.  byte myArray[13];
3.
4.  void setup()
5.  {
6.    Serial.begin(9600);
7.  }
8.
9.  void loop()
10. {
11.   myArray[0] = 0xc0; // start character
12.   int index = 1;
13.   // go through the analog pins one by one
14.   for(int i = 0; i < 6; i++) {
15.     unsigned int knob = analogRead(i);
16.     myArray[index++] = knob & 0x007f;
17.     myArray[index++] = knob >> 7;
18.   }
19.   Serial.write(myArray, 13);
20. }
```

We have explained almost everything we didn't already know for this code. Only in line 15 we see a new data type, which is `unsigned int`. This is actually an `int` without a sign, meaning it doesn't store negative numbers. Quoting from Arduino's reference: "On the Uno and other ATMEGA based boards, unsigned ints (unsigned integers) are the same as ints in that they store a 2 byte value. Instead of storing negative numbers however they only store positive values, yielding a useful range of 0 to 65,535 ((2^16) – 1)."
Since a byte can only be between zero and 255, there's no point in trying to transfer a negative number, so `unsigned int` is maybe a more logical data type to use. Still, the analog pin resolution of Arduino UNO is 10-bit, which is well within the positive range of a signed `int`, but it's good practice if we want to use other boards with greater resolution.

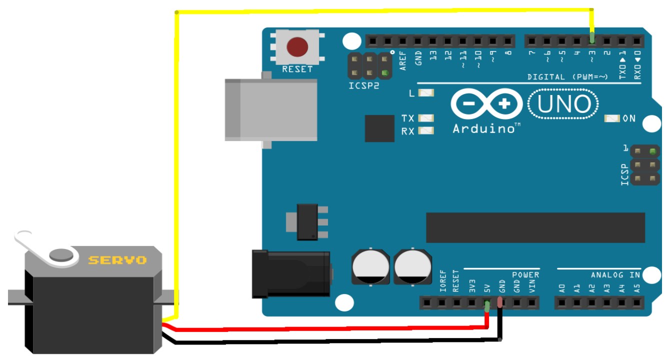
The Pd patch for this project follows the same philosophy with the patch of the third project, but we need to modify it slightly in order to read six pins instead of one. It's the following:



Notice the argument given to [repack], it's now 12. Remember this argument must be one less than the size of the array that is transferred from Arduino to Pd (the first value of the array is absorbed by [select 192]). Below the lower [spigot] we have [unpack f f f f f f f f f f f f] that unpacks 12 numbers and reassembles them the same way it did in the third project. Notice also that [t f b] in only at the very beginning of [unpack], which will actually be the last value to be output. Only after we have unpacked all values should we close the upper [spigot].
One last thing is the arguments given to [comport]. They are the serial port number, in case you know it and want to open it as soon as you open your patch, and the baud rate. The baud rate has to be the same with the Arduino's baud rate, otherwise communication between the computer and Arduino won't be possible[11].

Project 7 - servo motors

In this project we'll see how to control servo motors from Pd. This project's circuit is very simple, just note that we need a PWM pin to control the motor. Here it is:



Arduino has a library for controlling servo motors, which is `Servo.h` and this library has to be imported to our code in order for our sketch to work. To import a library to Arduino you need to write the following line at the beginning of your code:

11 Check the lower right pop up menu on Arduino's IDE Serial monitor to see available baud rates

```
#include <Servo.h>
```

This will tell the compiler to look for the specified header file (files with the .h extension are called header files) which header file should include functions which are not included in Arduino's core[12].
After we import the library we need to create an object of the Servo class. A class is a set of data, called a data structure, which we can call many times for many objects, where each object can have different values for each data of the structure. Think of it like an object in Pd. You can create many instances of [phasor~] where each one will have its own frequency and you can set a different phase for each. To create an object of the Servo class you need to write the following:

```
Servo servo;
```

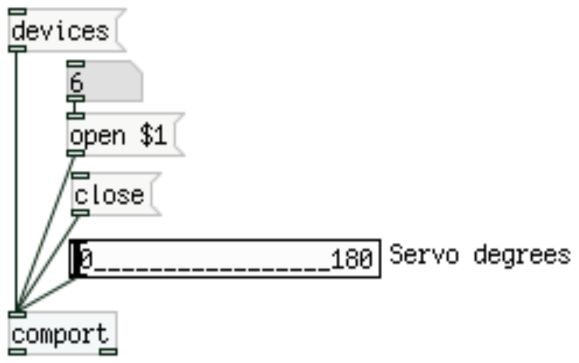
The name `servo` can actually be anything, like we can give any name to a variable, but since we'll be using servo motors, `servo` seems like a logical choice. We can distinguish it from the actual Servo class from its lower case first letter and from the fact that it's not color highlighted.
For this project we'll use two methods of the Servo class, attach and write. The first one assigns a pin to an object and the second one writes a value to that pin. The values are written in degrees from zero to 180. So, our code for one servo motor would be the following:

```
1.  // import the Servo library
2.  #include <Servo.h>
3.
4.  // create a Servo object
5.  Servo servo;
6.
7.  // variable to hold the PWM pin number
8.  int PWMpin = 3;
9.
10. void setup()
11. {
12.   servo.attach(PWMpin);
13.   Serial.begin(9600);
14. }
15.
16. void loop()
17. {
18.   while(Serial.available()){
19.     byte PWMbyte = Serial.read();
20.     servo.write(PWMbyte);
21.   }
22. }
```

In line 12 we call the attach method of the Servo class and give it the argument `PWMpin`, which is 3. And in line 20 we call the write method of the Servo class which takes one argument only, as we call it for the instance `servo`. This way Arduino knows which pin to write the value to, as we've attached that pin to the specific instance, so we don't need to specify it again.

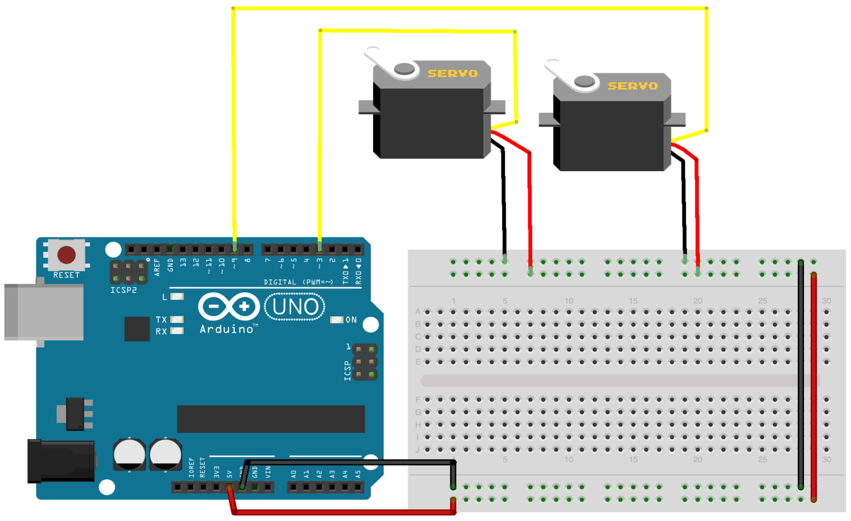
12 Note there's no semicolon at the end of the line

The Pd patch is the following:



It's almost identical to the patch of the fourth project, the analog output project. Only, this time, instead of sending values from zero to 255, we send values from zero to 180, as this is the range of the Servo library.

Lets work a bit more on this project. What if we want to use two servo motors and have them move independently? We'll have to combine the code above with the code we wrote for project 5, where we were controlling two digital pins separately. The circuit of this project is the following:



For this project though, we'll have to create two instances of the Servo object. We will also be sending a message from Pd in the form [print $1a$2b(where the letters function as separators of data so that incoming values can be diffused and stored in the correct variables. So our code will take the following form:

```
// import the Servo library
#include <Servo.h>
// define the number of servos used
#define NUM_OF_SERVOS 2

// create a Servo object array
Servo servos[NUM_OF_SERVOS];

// array to hold the PWM pin numbers
int PWMpins[NUM_OF_SERVOS] = { 3, 9 };

```

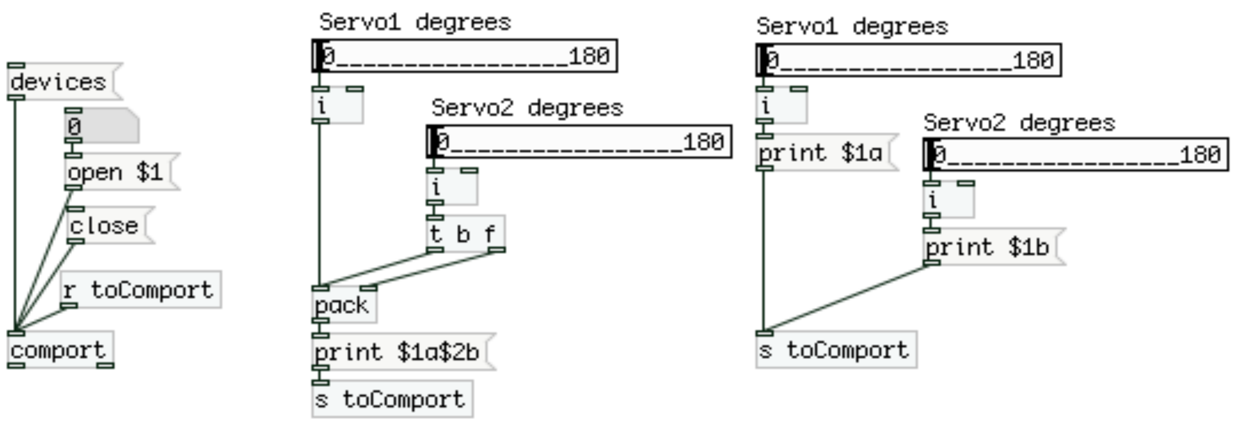
```
12. // array to hold the motors degree values
13. byte PWMbyte[NUM_OF_SERVOS];
14.
15. // variable to hold and assemble incoming data
16. int temporary;
17.
18. void setup()
19. {
20.   // attach pins to servo objects with a for loop
21.   for(int i = 0; i < NUM_OF_SERVOS; i++)
22.     servos[i].attach(PWMpins[i]);
23.   Serial.begin(9600);
24. }
25.
26. void loop()
27. {
28.    while(Serial.available()){
29.      byte inByte = Serial.read();
30.      if((inByte >= '0') && (inByte <= '9'))
31.        temporary = 10 * temporary + inByte - '0';
32.      else if((inByte >= 'a') && (inByte <= 'z')){
33.        // check whether we're exceeding array size
34.        if(inByte - 'a' >= NUM_OF_SERVOS) break;
35.        // if we're fine, write byte to corresponding array element
36.        PWMbyte[inByte - 'a'] = temporary;
37.        temporary = 0;
38.      }
39.      // write to servo pins with a for loop
40.      for(int i = 0; i < NUM_OF_SERVOS; i++)
41.        servos[i].write(PWMbyte[i]);
42.    }
43. }
```

In line 4 we `define` a so called macro which is highlighted by writing it in capital letters[13]. In this example we define the number of servo motors we're going to use, which is two. When we `define` a macro this way we don't need to end the line with a semicolon, like we `#include` libraries. Setting the number of motors used this way is quite intuitive and will be helpful in the rest of our code.
In line 7 we declare a `Servo` object array. We do that just like declaring an array of any standard data type (`int`, `byte`, etc.). Once a data structure has been created, in many cases it behaves like standard data types. Note also that we're using our macro to determine the size of the array. We'll use it in quite some places in our code.
In line 10 we declare and initialize an array to hold the pin numbers where we'll `attach` the servos. Initializing the array means to assign values to its elements, in this case 3 and 9, between the curly brackets. In line 13 we declare (but we don't initialize) an array to hold the values to be written to the servo motors.
In line 21 we run a `for` loop and use our `Servo` object array and the array that holds the pin numbers to `attach` a pin to each `Servo` object. This part of the code is very similar to the first version of this project, where we use one servo motor only. Instead, though, of writing `servo.attach(PWMpin)`, we use arrays for both `servo` and `PWMpin` variables. This loop also demonstrates how an object works in the

13 Capital letters are not necessary, it's more of a traditional way of defining macros

Arduino language, since we can call any function of the Servo class, for one object only, without any conflict between the created objects.
In our loop function we run a while loop as usual to check our input. In line 30 we check whether the data is a number and if it is, we store it in temporary. In line 32 we check whether the data is a letter and if it is, in line 36, we assign the value stored in temporary to the corresponding element of the PWMbyte array. We achieve this by subtracting 'a' from inByte. If inByte is 'a', ASCII 97, and we subtract 'a', we'll get 0, which is the first element of the array. If it's 'b', we'll get a 1, etc. This way we don't need to change anything in this part of the code, for as many servo motors we'll use (Arduino has only six PWM pins, so for sure we're not going to run out of letters). It will work only by changing the value of our NUM_OF_SERVOS macro.
For safety reasons, in line 34 we check the incoming letter to make sure it's not exceeding our array size, as this would very likely cause a crash. We do this by checking if inByte - 'a' is greater than or equal to our macro (which states the number of servos and the size of all arrays)[14]. If it's true, we call the break control structure, which will exit our while loop and save the program from crashing. Quoting from Arduino's reference "**break** is used to exit from a **do**, **for**, or **while** loop, bypassing the normal loop condition." Notice that we've written if's code immediately after it, on the same line. This syntax is also valid, but we usually avoid it for clarity reasons. In this case though, the code being one word only, we can as well write it this way.
We've already explained line 36, so we jump to line 40. In this line we run a for loop which writes the values stored in the PWMbyte array, to the corresponding Servo objects, much like we attached pins to the servos in line 21. We include this loop in the while loop so that we call the write function only in case we receive input, saving Arduino from doing unnecessary work[15]. Calling the write function for one servo only, seems to give unexpected results and jerky movement, so we need to call it for every Servo object.
We can see that this code is very flexible as we can use up to six servo motors (as many as the PWM pins of Arduino), and all we need to do is change our macro (and the circuit of course), plus slightly modify our Pd patch.

The Pd patch for this code is this:



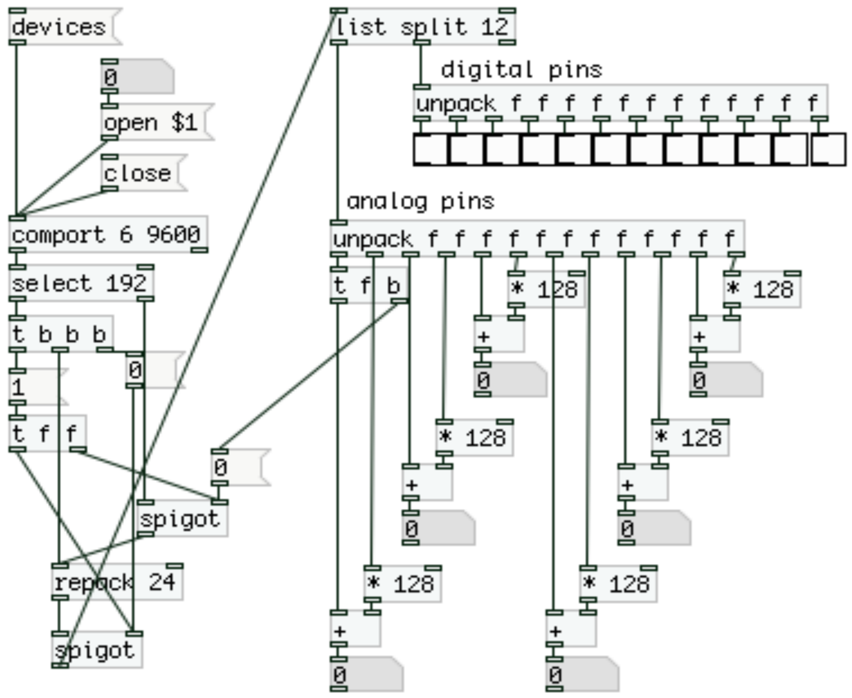
14 NUM_OF_SERVOS is 2, but our arrays start counting from 0, so if inByte - 'a' equals 2, we're already exceeding the size of our arrays, hence we're testing also for equality
15 If it were outside the while loop, it would be inside the main loop function and the write function would be called in every single loop of the program

The middle part is very similar to project's 5 patch, where we were controlling two digital pin separately. This time we're not using an oscillator, nor are we collecting data from Pd, using [r pd]. We only use two sliders with their range set to 0 – 180. We send the values to [i] first to make sure only the integral part is sent to Arduino. For some reason, when controlling more than one PWM pins on the Arduino, sending floats will yield unwanted results, so we need to send integers.
Note that you can also send information for one servo only, like we do in the right part of the patch, and it will still work. Even if you send a message like "print $1c" which would be for a third servo, Arduino won't crash, since we do a safety check of the letters, you just won't see anything happening.
We could use more servos to make a small robot that dances to the music Pd plays...

Project 8 – analog and digital input
In this project we'll combine project 6 with project 2. We'll use multiple potentiometers and multiple push buttons. We don't need a circuit image as the potentiometer circuitry is identical to project's 6 and the button circuitry follows the same line. So we'll jump straight to the code. By now it should be rather obvious as to how to implement this project. Since we want to transfer several values over to Pd, we'll need to use an array and use the Serial.write function. We've already used this technique for the six analog inputs and we can apply it to the digital inputs as well. The code is the following:

```
// declare the array to hold the values to be transferred to Pd
// the size of the array is the number of buttons + two times the
// number of potentiometers + 1, for the beginning of the array
byte myArray[25];

void setup()
{
  for(int i = 2; i < 14; i++)
    pinMode(i, INPUT);
  Serial.begin(9600);
}

void loop()
{
  myArray[0] = 0xc0; // start character
  int index = 1;
  // go through the analog pins one by one
  for(int i = 0; i < 6; i++) {
    unsigned int knob = analogRead(i);
    myArray[index++] = knob & 0x007f;
    myArray[index++] = knob >> 7;
  }
  // go through the digital pins one by one
  for(int i = 2; i < 14; i++)
    myArray[index++] = digitalRead(i);

  Serial.write(myArray, 25);
}
```

As described in the comment in the beginning of the code, the array needs to hold one value for the beginning of the data stream, two values for each analog pin (the reason for this has been explained in project 3) and one value for each digital pin (this will be either 1 or 0, so there's no way any of them will ever reach the value that denotes the beginning of that data stream). We'll be using all six analog and all twelve digital pins. That makes a total of 25.
Apart from that, the only new thing in this code is that once we're done with the loop that goes through the analog pins and sets the corresponding values to the corresponding elements of the array, we run another `for` loop that goes through the digital pins, this time calling `digitalRead` (rather obvious). We use the post increment technique again to assign the values read from the digital pins to the correct elements of `myArray`. Once both loops have finished, we've stored all values we want to read to `myArray` and we can call the `Serial.write` function to transfer the stored data to Pd.
The Pd patch is very similar to project's 6 patch, but not identical. It is an extension as project's 6 patch was an extension of project's 2 patch. We could omit including it here, but for the sake of clarity we won't just unpack 24 elements of the list we'll be receiving, but we'll be splitting the list into two, splitting the analog from the digital values we receive from Arduino. We'll also use toggles for the digital values as we did in project 2, as this is more intuitive for their projection. You can see the patch below:



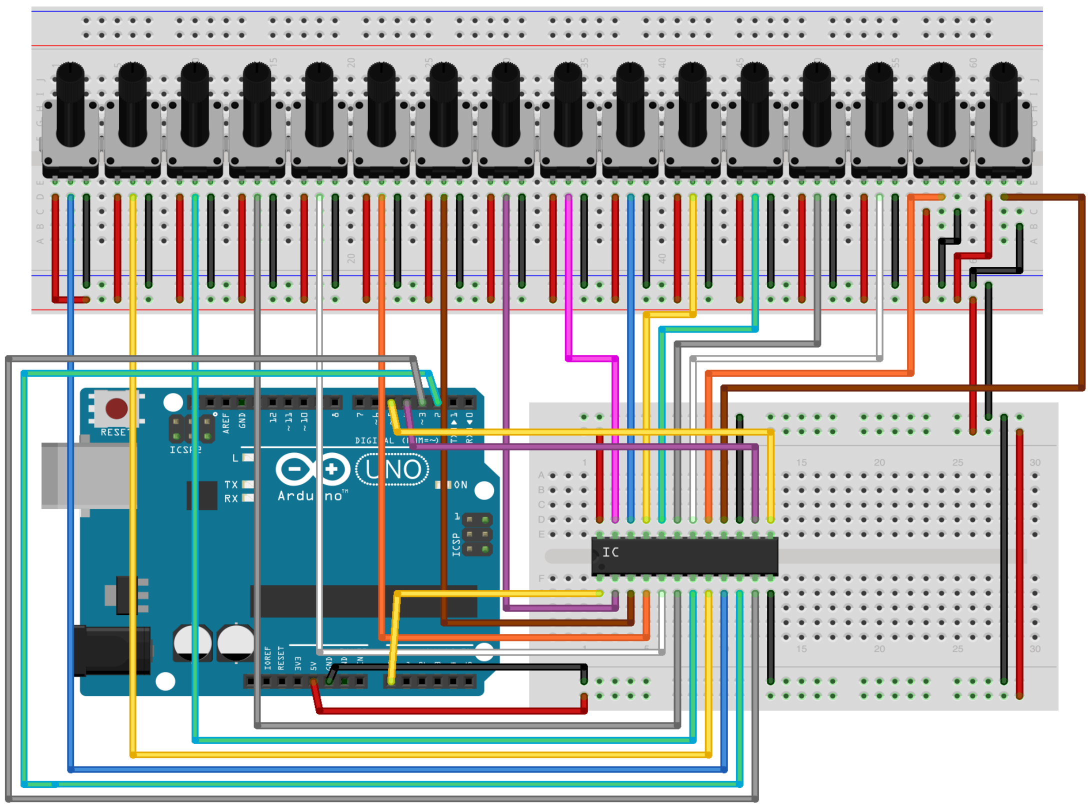
In order to split the incoming list and make things clearer we'll use [list split]. As a reminder we should point out that the argument to [repack] is the size of the transferred array minus one (the first element is being absorbed by [select 192]). Once [repack 24] collects 24 values it outputs them and [list split 12] receives this list and splits it into two[16]. The first half of the list is the analog pins that have been disassembled in order to be properly transferred and they are being reassembled in order to retrieve their 10-bit resolution.
The second half of the list is the digital pins. There's no special action needed here as

16 Check [list split]'s help patch to see how it works

these values will never reach 192, so we can just unpack these tweve values and project them with toggles.
Using this technique we can build a controller with six knobs (or other analog sensors) with a 10-bit resolution and twelve switches. If we want to include LEDs or other digital output, we can combine earlier projects to achieve this.

Project 9 – multiplexing

In this project we'll use a sixteen channel multiplexer to increase the number of available analog pins. With one such multiplexer we can get sixteen analog pins out of one analog pin of the Arduino! Of course this chip doesn't come so inexpensive (as far as the pins it uses, not its price), it also needs four digital pins, but that is not a problem as we'll see later on.
We'll be using the CD74HC4067 16-Channel Analog Multiplexer/Demultiplexer. This chip has sixteen inputs where each one can be routed to its output, but only one at a time. Which input of the multiplexer will be output to the Arduino is set with its four digital pins. These pins can have sixteen different combinations (two to the fourth power yields sixteen), so setting `HIGH` or `LOW` these pins will control which analog pin we'll be reading.
The circuit for this project might look a bit ugly on a breadboard and a bit difficult to understand its connections so be careful when you build it[17].



The wiring for the potentiometers is the same as with the earlier projects, one leg goes to 5V, one to ground and the middle one to an analog pin of the multiplexer. The chip has 24 pins, 16 of which are connected to the knobs. The first pin (the leftmost from the lower side) is the chip's output, which goes to an analog input in the Arduino. The last pin (the leftmost from the upper side) goes to 5V, pin 12 (the rightmost from the lower side) goes to ground, as well as pin 15 (three from right on the upper side). The remaining four pins are the digital pins that control the routing

17 The actual chip is thicker than the one in the image, but Fritzing didn't have such a component

of the analog pins. These are pins 10, 11, 14 and 13, with this order. Check the chip's data sheet for more information. You can get it here.

The code has many similarities with project's 6 and 8 code, but there are a few new thighs we'll be using. Here it is:

```
1.  // Chip's control pins
2.  #define CONTROL0 5
3.  #define CONTROL1 4
4.  #define CONTROL2 3
5.  #define CONTROL3 2
6.
7.  #define NUM_OF_MUX 1
8.
9.  // transfer array size, must be of const type
10. const byte numOfTransferData = ((NUM_OF_MUX * 16) *2) + 1;
11.
12. // buffer to hold data to be transferred over serial
13. byte muxArray[numOfTransferData];
14.
15. void setup()
16. {
17.   pinMode(CONTROL0, OUTPUT);
18.   pinMode(CONTROL1, OUTPUT);
19.   pinMode(CONTROL2, OUTPUT);
20.   pinMode(CONTROL3, OUTPUT);
21.   Serial.begin(115200);
22. }
23.
24. void loop()
25. {
26.   muxArray[0] = 0xc0; // denote beginning of data
27.   int index = 1;
28.   // loop to read all chips
29.   for(int i = 0; i < NUM_OF_MUX; i++){
30.     // this loop creates a 4bit binary number
31.     // that goes through the multiplexer pins sequentially
32.     for(int j = 0; j < 16; j++){
33.       digitalWrite(CONTROL0, (j&15)>>3);
34.       digitalWrite(CONTROL1, (j&7)>>2);
35.       digitalWrite(CONTROL2, (j&3)>>1);
36.       digitalWrite(CONTROL3, (j&1));
37.       unsigned int knob = analogRead(i);
38.       muxArray[index++] = knob & 0x007f;
39.       muxArray[index++] = knob >> 7;
40.     }
41.   }
42.     Serial.write(muxArray, numOfTransferData);
43. }
```

In lines 2 to 5 we define macros for the chip's control pins. In line 7 we define a macro that will hold the number of multiplexers we'll be using. In this particular

project we'll be using only one chip, but if we want to expand it, we'll only need to change this number instead of writing more code. In line 10 we define a variable of the type `const byte` that will hold the size of the array. An array must have a fixed size, therefore its size should be of a `const` type. There are techniques for changing the size of an array but it's very rarely used so we won't cover it here.
In the setup function we set all the control pins as outputs. Mind that these are the Arduino pins that will be sending data to the chip's pins. The chip's control pins are input pins that receive either a `HIGH` or `LOW` state. After that we begin the serial communication, but this time with a much higher baud rate, 115,200. This is the highest baud rate of Arduino and it's many times a desirable rate in order to have a fast response from it.
In line 29 we begin a `for` loop that will go through all the chips. If we were to use more than one chip, this loop would repeat for all of the chips, assigning each value to the transfer array, sequentially.
In line 32 we start another `for` loop which sets the control pins in a such a state that they create a 4-bit binary number that goes from 0 to 15. This way we're running through the sixteen potentiometers of the chip, one by one. Every time we're reading a new potentiometer, we're storing its value to the next element of the transfer array.
Finally, in line 42 we call the `Serial.write` function to transfer the data to Pd.
If you want to grasp the workings of the control pin loop (the code between lines 32 and 36), create a patch in Pd using [&] and [>>] to see how these four bitwise operations work together.

The patch for this project is the same as project's 6, only we'll be reading 16 potentiometers, so you'll need to modify it accordingly.

To conclude this project, if you want to use more chips, each chip will be wired to one analog pin of the Arduino, but they will all use the same digital pins, so for four digital pins and six analog, you can get up to 96 analog pins! Or to make it even more extreme, you can use one multiplexer to control 16 multiplexers, for one analog pin, but you'll need four digital pins for the main multiplexer and another four for all the other multiplexers. This circuit though (along with a functioning code) can yield up to 256 analog pins!

Project 10 – shift registers

10a – output shift registers

In this project we'll use shift registers to extend Arduino's digital pins, both for input and output. We could use multiplexers for this purpose (at least for input), but shift registers are by far a more effective technique.
We'll start with output shift registers as, traditionally, electronics projects start with lighting up LEDs.

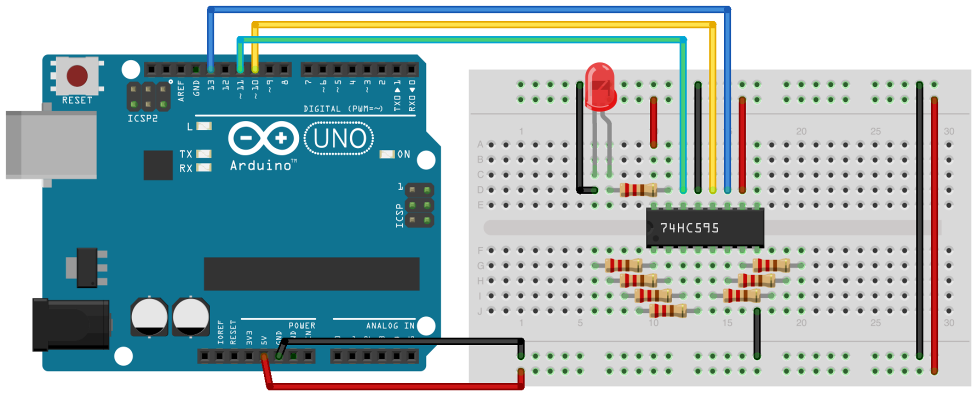
For this project we'll use the 74HC595 chip, a widely used, well documented and supported shift register. There are two main advantages shift registers have over multiplexers, when it comes to digital pins. The first advantage is that these chips collect their data serially, but output it in parallel, so it can light up up to eight LEDs (it's an 8-channel chip) simultaneously, and not alternately, like the multiplexer does, which creates the illusion of dimming. The second advantage is that these chips can be daisy chained, meaning that one chip passes data to the next in the chain, without requiring more pins from the micro-controller. This way you can extend the digital pins of the Arduino to a great extent (LED cubes with 1000 or so LEDs are being built with shift registers and only one Arduino!).

An 8-channel shift register collects a byte which represents the state of its output pins. If the byte is 0x00, all pins will be LOW. If this byte is 0xFF, then all its pins will be HIGH. If the byte is 0x03, the first two pins will be HIGH and the rest LOW[18], and so on. For this reason some bitwise operations will come in handy. Also, we'll be using the SPI library instead of Arduino's shiftIn and shiftOut functions, as it is much faster and easier to grasp.

Say that all pins of the chip are initialized `LOW`, and we want to turn on the third LED only. In order for our program to be intuitive, we'll need to write code that will accept the number 3 for the pin and the number 1 for its state (or 0 if we want to turn it off). Say that we send the message [print 3l1s(where l stands for LED and s for state. In this case what we need to do is assign the number 4 to the byte of the chip, as four in 8-bit binary is 00000100. This number will turn the third pin `HIGH` and will keep the rest `LOW`.
If, afterwards, we want to turn the fifth LED on as well, without turning the third LED off, we need to assign the number 20 to the chip's byte, as 20 in 8-bit binary is 00010100. But how can we manage this? Thankfully, the Arduino language provides a function that writes a value to a specific bit of a byte, `bitWrite`. This function takes three arguments, the byte to write the bit to, the specific bit of the byte, and lastly the value to write to that bit. For example, if we initialize a byte to the value 0 and want to write the value 1 to its third byte, then we would use this function like this:

```
bitWrite(the_byte_variable, 3, 1);
```

This would yield the number 4, binary 00000100, which would essentially turn on the third LED of the shift register. Before we proceed with the code lets see the circuit[19]:



Now we can start writing code. In this project we'll introduce a new control structure, the `switch` / `case` statement. Its concept is similar to the `if` statement, but it's expressed a bit differently. It's syntax is the following:

```
switch (var) {
     case 1:
        //do something when var equals 1
        break;
```

18 The 0x prefix indicates a hexadecimal value, where FF is 255, the full range of a byte. Hexadecimal values are used a lot with shift registers as they are convenient when bytes are expressed
19 For clarity only one LED is included in the image, but the actual circuit includes eight. Their long leg connects to each 220Ohm resistor and the short to ground

```
      case 2:
        //do something when var equals 2
        break;
      default:
        // if nothing else matches, do the default
        // default is optional
  }
```

This control structure takes a variable as an argument and it then tests to see if it equals any of the cases below. In the example above[20] we give it the variable `var` and it tests if `var` equals 1 or 2. If it equals 1, then the code after the first colon will be executed and the control structure will end. If it equals 2, then the code after the second colon will be executed, and if it equals none of the above, the code after `default` will be executed. Note that `default` is optional, as the `else if` and `else` statements are in an `if` control structure. Also note that at the end of each `case` statement we need to call `break` as we did in project 7 (without it, even if the test variable equals one of the statements, the `switch` / `case` structure will keep on going until all cases have been tested).

Now that we've covered the `switch` / `case` statement, we can move on to write the full code.

```
#include <SPI.h>

const byte latch = 10;

// set here the number of chips
const byte numberOfChips = 1;
// declare and initialize LED array
byte LEDdata[numberOfChips] = { 0 };

// variable to hold incoming data temporarily
int temporary;
// variables for data diffusion
int chip;
byte pin, state;

// function that calls SPI.transfer to transfer data to chips
void refreshLEDs()
{
  digitalWrite(latch, LOW);
  for(int i = numberOfChips - 1; i >= 0; i--)
    SPI.transfer(LEDdata[i]);
  digitalWrite(latch, HIGH);
}

void setup()
{
  Serial.begin(115200);
  SPI.begin();
  refreshLEDs();
```

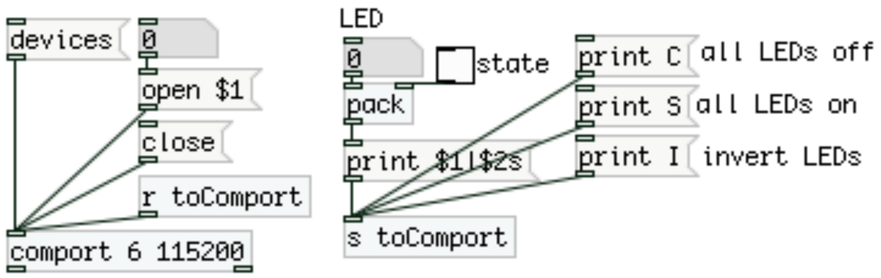
20 Taken from Arduino's website

```
30. }
31.
32. void loop()
33. {
34.   while(Serial.available()){
35.     byte inByte = Serial.read();
36.     if((inByte >= '0') && (inByte <= '9'))
37.       temporary = 10 * temporary + inByte - '0';
38.     else{
39.       switch(inByte){
40.         // set byte to store to LEDdata array
41.         case 'l':
42.           // subtract one so that the Pd patch is more intuitive
43.           temporary -= 1;
44.           // divide by 8 to work out which chip
45.           chip = temporary / 8;
46.           // remainder is pin
47.           pin = temporary % 8;
48.           // reset temporary
49.           temporary = 0;
50.           break;
51.         // set state of LED and write bit to byte
52.         case 's':
53.           state = temporary;
54.           // write bit to array byte
55.           bitWrite(LEDdata[chip], pin, state);
56.           // reset temporary
57.           temporary = 0;
58.           break;
59.         // C: clear all bits
60.         case 'C':
61.           for(int i = 0; i < numberOfChips; i++)
62.             LEDdata[i] = 0;
63.           break;
64.         // S: set all bits
65.         case 'S':
66.           for(int i = 0; i < numberOfChips; i++)
67.             LEDdata[i] = 0xFF;
68.            break;
69.         // I: invert all bits
70.         case 'I':
71.           for(int i = 0; i < numberOfChips; i++)
72.             LEDdata[i] ^= 0xFF;
73.           break;
74.        }
75.     }
76.     refreshLEDs();
77.   }
78. }
```

In line 1 we import the SPI library. In line 3 we assign the number 10 to the so called latch pin (we'll explain what it is in a bit). Notice line 14 where we declare two

variables of the same type in the same line, separated by a comma.
In lines 17 – 23 we declare a function of our own, which we'll call from within the loop function. This function sets the latch pin LOW and runs a for loop for as many times as the chips we're using (for now we're using only one, but changing the number in line 6 would suffice in order to use more), calling the SPI.transfer function with the corresponding byte of the current chip as an argument. This time the loop goes from the highest to the lowest value and decrements its variable i, by using the C symbol for decrementing, two minuses. After the loop has finished, the latch pin goes HIGH. When the latch pin goes HIGH (shift register's pin 12) the chip outputs all the data it has received.
In line 28 we call the SPI.begin function which starts all the SPI functions, much like Serial.begin, but without an argument. In line 36 we do the usual check to see whether the incoming data is a number and in line 39 the switch / case structure begins. Here we check whether the inByte variable equals one of several letters. If it equals 'l', we assign the value that came in the serial line to the pin variable and set the correct chip number. There are quite a few comments there so the code should be quite self-explanatory.
If inByte equals 's', we assign the value stored in temporary to the state variable and call the bitWrite function. The arguments to the function are the byte stored in the LEDdata array element, with the index stored in chip, the bit position which is stored in pin and the actual value to write to that bit which is stored in state. In both cases where inByte equals 'l' or 's' we reset temporary to 0 so we can start collecting values from the serial line from the start.
The rest three cases are copied from the tutorial sketch of Nick Gammon, where in the case of the letter 'C', we set all bytes of the LEDdata array to 0, in the case of 'S' we set them all to 255 (0xFF in hexadecimal), and in the case of 'I' we XOR[21] each byte with 0xFF. Notice that line 72 reads LEDdata[i] ^= 0xFF; which is equal to LEDdata[i] = LEDdata[i] ^ 0xFF; but the C programming language provides shortcuts for such expressions[22].
Finally, in line 76 we call our custom function, refreshLEDs() which we've declared earlier in our code. As with the servo project, we're calling this function inside the while loop so Arduino doesn't have to do unnecessary work.
One last thing to note is that you can use more chips by daisy chaining them. The circuit is exactly the same for as many chips you want to use, only pin 14 of the second chip (not Arduino), is connected to pin 9 of the first chip, instead of connecting to Arduino's pin 11 (pin 14 of the third to pin 9 of the second, etc.). This way the data goes through from one chip to the other. Check the chip's specs here.

We should presume that the data sent from Pd should be in the form “print $1l$2s”, or “print C” (or the other capital letters). So the patch will be the following:

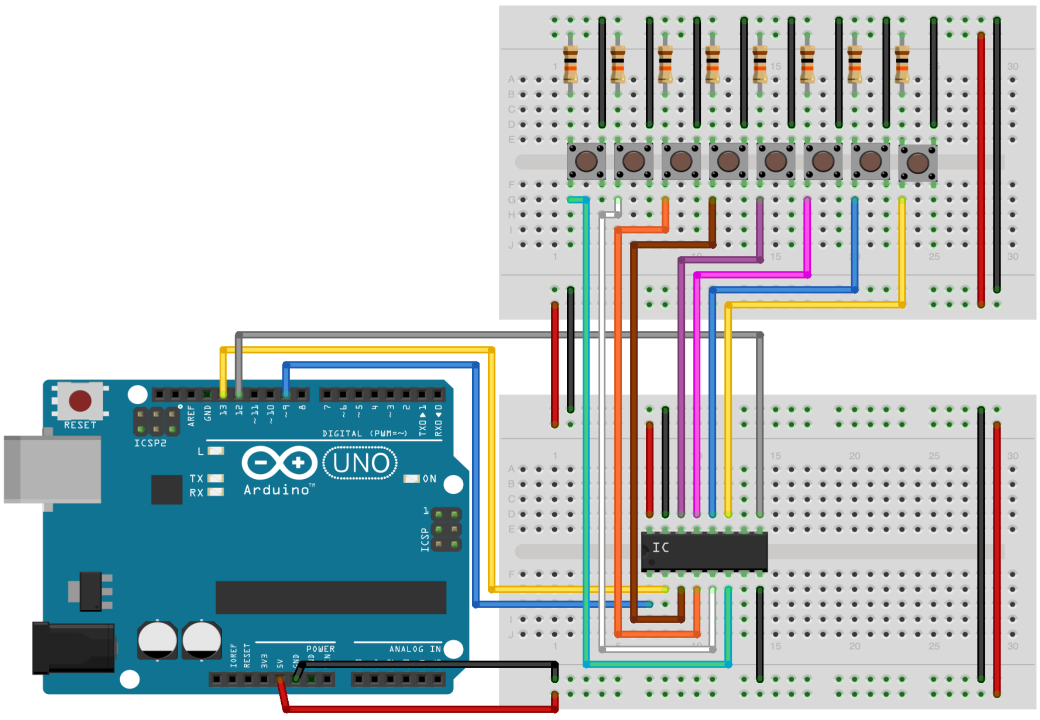


21 Exclusive OR, is a bitwise operation where two values are being tested and if their bits are equal (e.g. both 1 or 0) they are set to 0 otherwise they are set to 1. Check Arduino's reference for more information
22 Similar shortcuts exist for other operations like +=, -=, *=, /=, %=, &=, |=

An interesting application could be a digital VU meter which takes as input the values sent to [dac~], scaled and mapped to the appropriate range. For this project you might find other methods for setting the appropriate value to each byte, more useful, like using the pow function which raises a number to a power. I leave it to the reader to find out how such a method could be utilized.

10b – input shift registers

Continuing with the shift registers, we'll now deal with input, using the 74HC165, the input equivalent of the chip used in the first version of this project. This chip works in the other direction than the previous one, meaning it collects its data from its digital input pins in parallel and outputs them to Arduino serially. Arduino then collects one byte per chip which represents the state of each input pin of the chip, in a similar fashion to the chip used above. Lets first check the circuit for this project:



If you take a closer look you'll see that in contrary to project 2 (digital input), we use pull-up resistors and not pull-down. This means that one leg of each button is connected to 5V via a 10k resistor and not to ground. The result of this is an inversion of the readings of digital pins, where, when a button is pushed, Arduino will read LOW and the other way around. This is quite counter intuitive, but it is said that pull-up resistors are a more stable circuit, so this is what we'll use for this project.

The code for the input shift register is smaller and easier than that of the output shift register. Here it is:

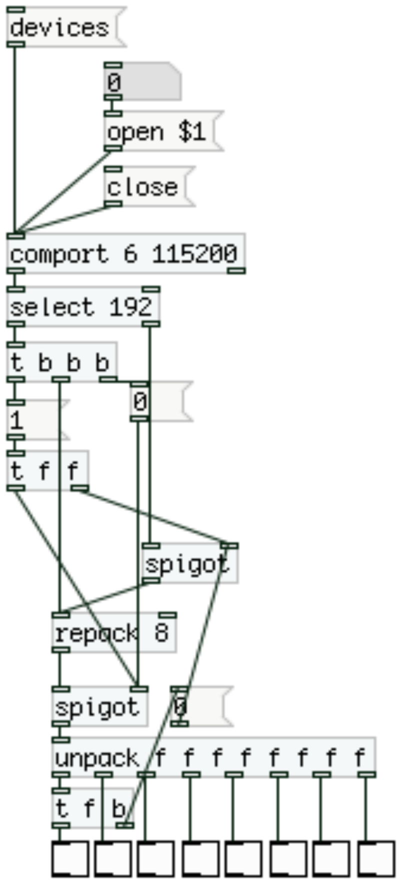
```
#include <SPI.h>

const byte latch = 9;

// set here the number of chips
const byte numberOfChips = 1;
// array to get bytes from SPI
```

```
8.  byte switchData[numberOfChips] = { 0 };
9.  // array and size of data to be transferred to Pd
10. const byte numOfData = (numberOfChips * 8) + 1;
11. byte transferData[numOfData];
12.
13. // function that calls SPI.transfer to transfer data from chips
14. void refreshSwitches()
15. {
16.   digitalWrite(latch, LOW);
17.   digitalWrite(latch, HIGH);
18.   for(int i = 0; i < numberOfChips; i++)
19.     switchData[i] = SPI.transfer(0);
20. }
21.
22. void setup()
23. {
24.   pinMode(latch, OUTPUT);
25.   digitalWrite(latch, LOW);
26.   Serial.begin(115200);
27.   SPI.begin();
28. }
29.
30. void loop()
31. {
32.   transferData[0] = 0xc0;
33.   int index = 1;
34.   refreshSwitches();
35.   for(int i = 0; i < numberOfChips; i++){
36.     for(int j = 0; j < 8; j++)
37.       transferData[index++] = bitRead(switchData[i], j);
38.   }
39.   Serial.write(transferData, numOfData);
40. }
```

In line 3 we declare pin 9 for the `latch` pin of the chip as a constant. This is because pin 10 on the Arduino is the so called Slave Select pin, which is a dedicated pin for the SPI library and it's used only for output chips. And since it's a dedicated pin for this purpose, we didn't need to set it as `OUTPUT` and initialize its state in our `setup` function, in the previous example. In this example though we need to do that, and we do it in lines 24 and 25, because pin 9 is not dedicated to something.
In line 8 we declare and initialize an array that will store bytes that come in from the shift registers. But in lines 10 and 11 we declare another array that will store the data that will be transferred to Pd. We'll see in a bit why we need to do that.
Lines 14 to 20 include our custom function `refresfSwitches()` which calls `SPI.transfer`. There are a few differences with the custom function we wrote in the previous example. First of all, we set the `latch` pin `LOW` and then `HIGH` immediately. This is because we don't need to send stored data to the `SPI.transfer` function which will later transfer that over to the chips, but the other way around, we need to collect data with the `SPI.transfer` function which we will store inside the Arduino. Once the `latch` goes high, data will start coming in from the chips serially. As this data comes in, we run a `for` loop and assign the current value `SPI.transfer` collects to the current byte of the `switchData` array. Notice the argument passed to

SPI.transfer which is 0, instead of an already stored value as we did previously. This argument tells this function to collect incoming bytes rather than transfer outgoing ones.
In our loop function we first assign the value 192 (c0 in hexadecimal) to the first element of the data transfer array, like we've done with most projects that transfer data from Arduino to Pd.
In line 34 we call our refreshSwitches function to store one byte for each chip. Then we run two nested for loops, where the second one runs eight times (as many as the pins of each chip) for as many times as the chips we're using. Inside the second loop we store to an incrementing index of the transferData array, the value read at the current bit of the current byte of the array that stored data from the chips. We achieve that by calling the bitRead function, which is similar to bitWrite, only it takes two arguments instead of three, the byte to read a bit from, and the bit to read. This way we store the state of each pin of the chip to an element of the transferData array. Finally we transfer the stored data to Pd by calling Serial.write.

The corresponding Pd patch looks like this:



If you run this project you'll notice that when you don't press a button, the corresponding toggle in Pd is on, and vice versa. This is because we've used pull-up resistors as already mentioned. It should be fairly easy to reverse this state in Pd or Arduino, we'll leave that to the reader.

If you want to use more than one chip (you can use really lots of them without occupying more pins on the Arduino), you need to wire all of them the same way, only pin 10 of the first chip connects to pin 9 of the second (so the second doesn't connect to Arduino's pin 12), and pin 10 of the second to pin 9 of the third, and so on. The rest of the pins have identical connections. Check the chip's data sheet for more information. You can get it here.

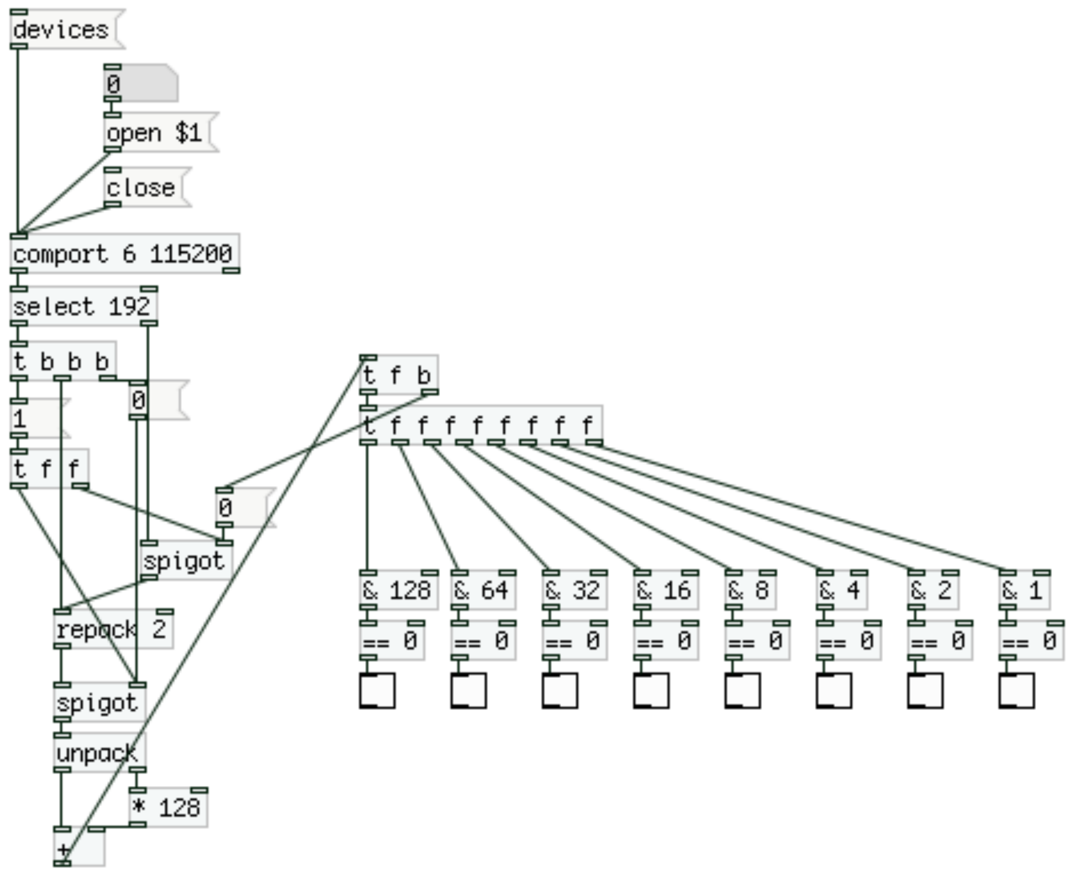
Before we finish with this project (and the tutorial itself), let's look at another way to code and patch it. If we want to use quite a lot of chips plus other stuff, we might reach a point where memory becomes a problem. Don't forget that Arduino has only 32 KB of memory. Since the `SPI.transfer` function stores one byte per chip, we can use that byte as is, instead of reading its bits one by one, as the latter will store a whole byte for the value of one bit.

We could modify the Arduino sketch slightly so it becomes like this:

```
#include <SPI.h>

const byte latch = 9;

// set here the number of chips
const byte numberOfChips = 1;
// array to get bytes from SPI
byte switchData[numberOfChips] = { 0 };
// array and size of data to be transferred to Pd
const byte numOfData = (numberOfChips * 2) + 1;
byte transferData[numOfData];

// function that calls SPI.transfer to transfer data from chips
void refreshSwitches()
{
  digitalWrite(latch, LOW);
  digitalWrite(latch, HIGH);
  for(int i = 0; i < numberOfChips; i++)
    switchData[i] = SPI.transfer(0);
}

void setup()
{
  pinMode(latch, OUTPUT);
  digitalWrite(latch, LOW);
  Serial.begin(115200);
  SPI.begin();
}

void loop()
{
  transferData[0] = 0xc0;
  int index = 1;
  refreshSwitches();
  for(int i = 0; i < numberOfChips; i++){
    transferData[index++] = switchData[i] & 0x7f;
    transferData[index++] = switchData[i] >> 7;
  }
  Serial.write(transferData, numOfData);
}
```

We won't analyze this code a lot, just note that we're not calling `bitRead` but we're saving the byte received from `SPI.transfer` to the `transferData` array. This way we save a lot of space. Though, since we'll be sending a byte for the state of all pins of the chip, we need to split it the same way we did with the analog readings in previous projects, as the transferred byte is very likely to take the same value as the byte that denotes the beginning of the data stream, 0xc0. So, in line 35, once we've called `refreshSwitches`, we run a `for` loop as many times as the number of chips, and assign each value of each element of the `switchData` array to two elements of the `transferData` array, much like we've done so far. Note that in line 36 we mask the byte with 0x7f, which is again 127, but has the size of a `byte` and not of an `int`. This code should be slightly faster as we omit calling an additional function, `bitRead`.

But we also need to modify the Pd patch as well, in order for it to work with this sketch. What we need to do is take the incoming byte and break it up to its bits. We can achieve this by using bitwise and boolean operations. We'll use the bitwise AND and the boolean test of equality[23]. In detail we'll bitwise AND the byte with the numbers 128, 64, 32, 16, 8, 4, 2 and 1. It's pretty easy to see the pattern here. After each bitwise AND we'll check if the output is equal to 0. Normally we would have to check whether it's not equal to 0, but since we're using pull-up resistors, checking for equality to 0 will also reverse the output and instead of getting a 0 when we press a button, we'll be getting a 1, which is what we essentially want.
Needless to say that we need to re-assemble the incoming byte to its full range, as we've been doing all along (and modify [repack]'s argument too). So our patch will now look like this:



23 Again, taken from rePatcher by Open Music Labs, and slightly modified

Conclusion

We have completed ten projects that build one on top of the other, extending our knowledge and understanding of the Arduino language bit by bit. The concepts used and explained in this tutorial are basic concepts of Arduino and programming in general. Using the techniques of this tutorial one can create lots of different projects, by combining the ones we've covered so far, by applying techniques described here in other projects etc. We could have developed more projects including DMX, or chips that extend the PWM pins of Arduino, like the TLC 5940, but the respective libraries include examples which by now should be easy to understand and modify to one's needs. Another popular project we didn't cover is solenoids, but its circuitry is beyond the scope of this tutorial. I hope this tutorial has given you insight and helped you get started with your own Arduino and Pd projects.

Alexandros Drymonitis
August 2014