CHAPTER 5

Hardware I/O Library

A feature that is common to most SBCs is the rows of expansion header pins they provide. Current version of the Raspberry Pi have 40 pins, arranged in two rows at one edge of the board. The CHIP has two rows of pins along both lateral sides of the board. Those pins make it possible to interface with external hardware peripherals, such as sensors and actuators. This chapter will introduce you to how this is done from within Processing.

Controlling hardware from Processing was traditionally done with the use of microcontrollers, such as *Arduino*. Your Processing sketch would open a serial connection with the microcontroller attached to the computer's USB port. Messages would then be sent back and forth to make the microcontroller act on the computer's behalf, and inform it about the results of sensor readings. Libraries such as *Firmata* are commonly used to aid with passing messages. On SBCs we have now the option of skipping this indirection, and directly using its own expansion header pins. (The option of using an Arduino as an auxiliary of course still remains. See also the section *Limitations of Hardware I/O* later in this chapter.)

This direct access to the very primitive, electrical fabric of computers is somewhat reminiscent of earlier generations of machines, that were less tightly integrated. The bulky parallel port interface, most commonly used to connecting printers in the 90ies, could so just as well be used to control a CNC machine. The serial port interface instead used as an ad-hoc network to very reliably synchronize multi-channel projection artworks.

Having this straight-forward way to connect to SBCs electrically sets them apart from most consumer electronics devices, and make them a *tool* for exploring and interacting with other parts of our electrified environment. The possibility of writing our own code sketches to live on an equal footing amogst the (e.g. *Smart Home*) appliances of our times is an exiting prospect.

Voltages

Many entry level Arduino boards, such as the *Arduino Uno*, use a supply voltage, or *VCC*, of 5 Volts. This voltage is as well also assumed in different schematics and tutorials found online. Most SBCs, in contrast, use a supply voltage of a mere 3.3 Volts. This is to be kept in mind when using them.

> Don't expose your expansion header pins to more than the supply voltage, which is 3.3 Volts in most cases. Pins can be made to be "5V tolerant", but this is not the case for the SBC models discussed here.

If you're following a schematic that uses 5 Volts, see if you can make it also work with 3.3. A simple voltage divider would, for example, work just as well with fewer Volts. Consult the datasheet of integrated circuits to see if they operate with 3.3 Volts as well.

Peripherals that insist on a higher voltage can be interfaced with using a *Bi-directional Logic Level Converter*, which are readily available in a pre-assembled form.

Digital Pins

Digital pins, also known as *GPIO pins*, work in a similar way as they do on *Arduino*. The following section will introduce the functions `pinMode`, `digitalWrite`, `digitalRead`, `attachInterrupt`, `releaseInterrupt`, `interrupt`, `noInterrupt`, as well as the `releasePin` function, which is unique to Processing's Hardware I/O library.

Using digitalWrite to control voltages

The `digitalWrite` function changes the output voltage of a pin from 0 Volts to 3.3 Volts and vice versa.

Lighting an LED

The following example will illuminate an attached LED when the user clicks inside the circle displayed on the screen.

Example 5-1.

```
import processing.io.*;
boolean ledOn = false;

void setup() {
  GPIO.pinMode(4, GPIO.OUTPUT);
  stroke(255);
}

void draw() {
  if (ledOn) {
    fill(255);
  } else {
    fill(204);
  }
  ellipse(50, 50, 75, 75);
}

void mousePressed() {
  if (sqrt(pow(mouseX-50, 2) + pow(mouseY-50, 2)) < 37.5) {
    // toggle boolean variable
    ledOn = !ledOn;
    GPIO.digitalWrite(4, ledOn);
  }
}
```

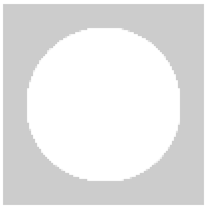

Figure 5-1. Display window after enabling the LED

This example shows how Processing's GPIO class is modeled after the familiar vocabulary from the Arduino project: The `pinMode` function together with the `GPIO.OUTPUT` argument makes the pin act as an output, i.e. being able to provide (source) or accept (sink) electrical current. The first argument contains the GPIO (*General-purpose input/output*) number of the pin, which is in our example GPIO 4.

Once the mouse is pressed inside the circle, the pin is switched from low (*0V*) to high (*3.3V*) with the `digitalWrite` function, which makes the LED light up. A future click, in which the second argument to the function is `false`, will switch the pin back to low (*0V*) and the LED will cease to be lit up.

The following illustration shows the wiring for this example on the Raspberry Pi:

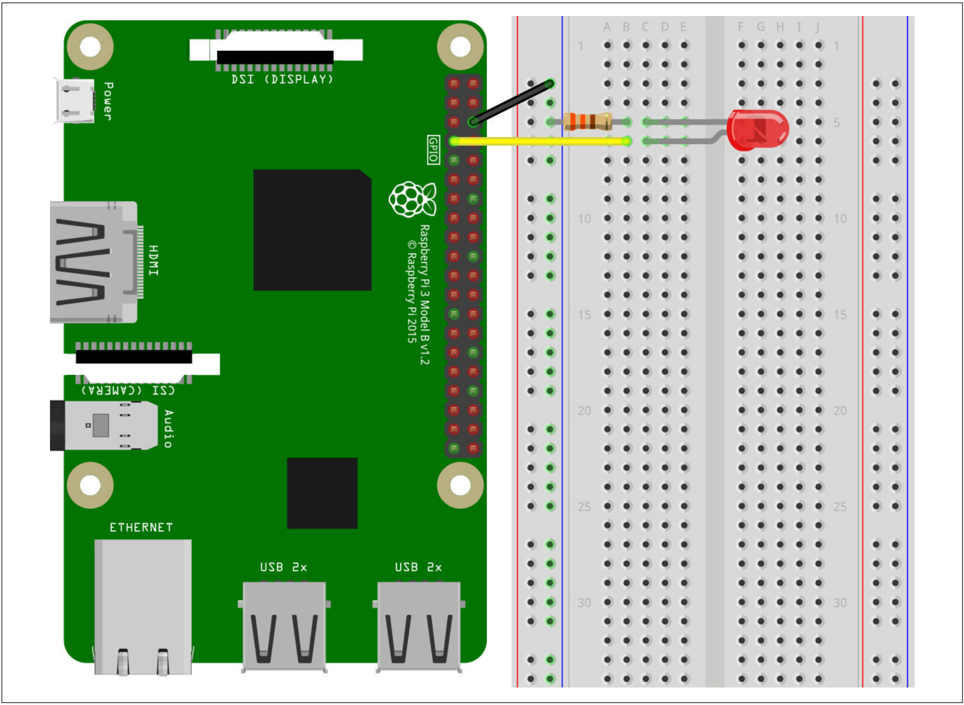

Figure 5-2. Wiring for Example 5-1

The longer leg of the LED, the anode, is connected to the pin on the expansion header. The shorter pin, the catode, is connected to Ground (*GND*) via a 330 Ohms resistor. This resistor is there to limit the flow of current somewhat, as to not cause damage to either the LED or the SBC.

We're connecting the LED to the forth pin on the left row, which according to normal numbering conventions would be pin 7. Why is this then *GPIO 4*, as mentioned above? This is because the GPIO lines are not assigned in order on the extension header. This book and most other literature will refer to the pin as the *GPIO pin 4*, rather than the (physical) *header pin 7*. Have a look at the full diagram on p. XXX to see which pin corresponds to which GPIO.

Instead of GPIO 4, this example could have also used a different pin, as long as the GPIO number in lines 4 and 21 would be changed accordingly. It is recommended to use those pins that not also serve a more specialized function, such as GPIO 14 and 15, which are also used as the serial port. Those pins might already be already in use by other parts of the system.

Besides turning on lights, the `digitalWrite` function is often used to control and activate more elaborate electrical circuits and devices.

Don't draw more than 16mA per pin from the Raspberry Pi, with a total of 50mA from all pins. Check the documentation of other SBCs for similar information. To switch larger loads make use of a MOSFET transistor whose gate is connected to the output pin via a resistor.

Lighting an LED by sinking current

Less commonly used, yet very versatile, is the ability to receive (sink) current though a pin. This is a different way of switching electical loads on or off using the `digital Write` function.

The previous example could also be wired this way:

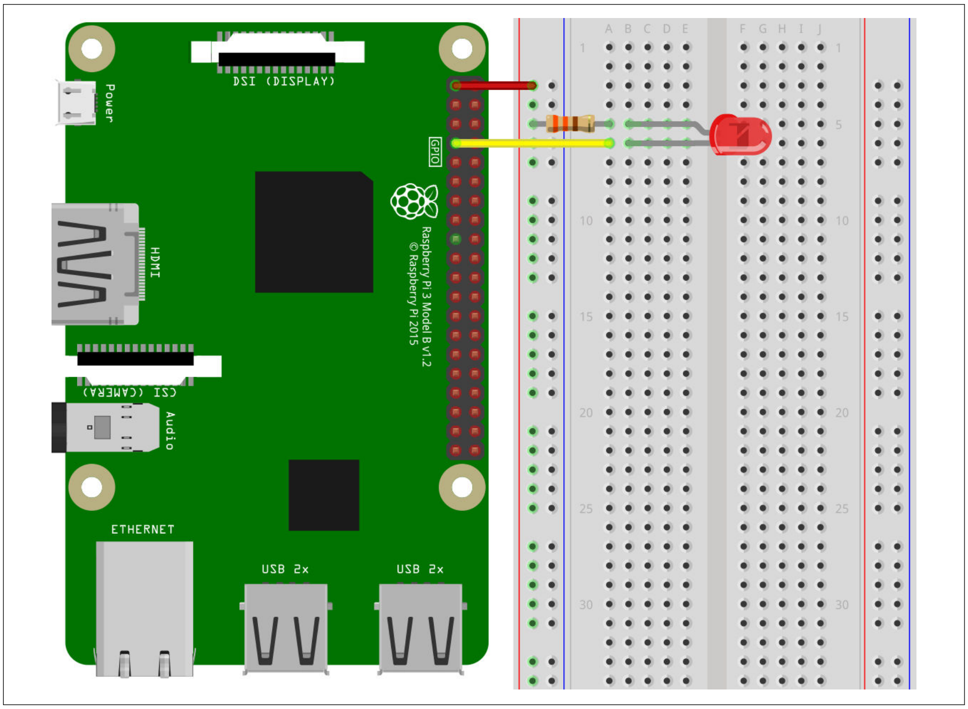

Figure 5-3. Wiring for Example 5-2

Here current flows from the 3.3V supply pin in the top left corner, through the 330 Ohm resistor and the LED, which now has its orientation flipped. If the pin is set to output and low (*0V*), it will accept (sink) the current coming from the LED, which will turn on. If the pin is set to output and high (*3.3V*), there is no difference in voltage (potential) between the LED and the pin, so no current will flow and the light will remain off. Similarly, if the pin is in input mode, which is the default, the pin does not complete the electrical circuit for the LED, and no current will flow either.

Using the pin as a sink inverts the logic of `digitalWrite`: passing `GPIO.LOW`, or `false`, as the second argument will make current flow, while `GPIO.HIGH`, or `true`, will make the action stop. The code of Example 5-1 can be used as-is, with the exception that the variable `ledOn` should be initialized with the value `true` to adequately describe the interaction between code and hardware.

Example 5-2.

```
import processing.io.*;
boolean ledOn = true;

void setup() {
  GPIO.pinMode(4, GPIO.OUTPUT);
  // LED will turn on as soon as the pin switches to
  // output and assumes a low state
  stroke(255);
}

void draw() {
  if (ledOn) {
    fill(255);
  } else {
    fill(204);
  }
  ellipse(50, 50, 75, 75);
}

void mousePressed() {
  if (sqrt(pow(mouseX-50, 2) + pow(mouseY-50, 2)) < 37.5) {
    // toggle boolean variable
    ledOn = !ledOn;
    GPIO.digitalWrite(4, ledOn);
  }
}
```

What, however, if we wanted the original behavior, in which the LED was off at the beginning of the sketch? One could change `setup` to read like this

```
void setup() {
  GPIO.pinMode(4, GPIO.OUTPUT);
  GPIO.digitalWrite(4, GPIO.HIGH);
  stroke(255);
}
```

This, however, leaves a minute window between the execution of one line and the next, in which the pin is driven high. Instead, we can set the initial voltage to be applied before the output is even enabled. This is done by calling `digitalWrite` ahead of `pinMode`.

```
void setup() {
  GPIO.digitalWrite(4, GPIO.HIGH);
  GPIO.pinMode(4, GPIO.OUTPUT);
  // pin will always remain high during setup
  stroke(255);
}
```

In practice, this is likely not a huge consideration in the case of an LED, but might be necessary if a short pulse emanating from the pin, however brief, might already cause an integrated circuit to behave in a way that is not wanted.

Don't sink more than 16mA per pin on the Raspberry Pi. Check the documentation of other SBCs for similar information.

Using digitalRead to read a pin's value

The `digitalRead` function returns whether a pin configured as input is low (*0V*) or high (*3.3V*). Unlike with Arduino, it is strictly necessary to call the `pinMode` function to declare each pin as an input before they can be used by this function.

Reacting to physical buttons

In Example 5-1 we reacted to the user pressing the mouse button over a drawn, virtual button on the screen. Instead, we can also have our sketch react to the press of a physical button or switch. Those come in different shapes and forms, from playfully-colored arcade buttons, to the sterile rigidity of a metal button you might expect to see inside an elevator, to tiny toggle switches once used on front panels of early computers to input data and instructions.

The underlying principle is the same: an electrical circuit gets either closed or severed by mechanical movement.

We can easily learn about the state of a button or switch by connecting it to a GPIO pin. But before, let's quickly discuss the concept of *electrical bias*: SBCs contain both internal *pull-up resistor* as *pull-down resistor* for each pin. Those are used to predispose the pin into being read back as either high or low when nothing else is connected.

On the Raspberry Pi, whether the pin is predisposed to being high or low varies from pin to pin. Different than on *Arduino*, Linux-based SBCs currently also don't provide a general way to enable and disable these *pull-up* and *pull-down resistors* as part of a software sketch.

A quick way to figuring out whether a pin has a *pull-up resistor* acting on it or a *pull-down resistor* is to read its value while nothing is attached to the pin:

Example 5-3.

```
// run with nothing attached to GPIO 4
import processing.io.*;

GPIO.pinMode(4, GPIO.INPUT);
if (GPIO.digitalRead(4) == GPIO.HIGH) {
  println("Pin is high");
} else {
  println("Pin is low");
}
```

For GPIO 4, running the sketch will output "`Pin is high`" in the Console. For other pins, and other SBCs, this might vary.

If you notice the reported value fluctuating this means that there is neither a *pull-up* nor a *pull-down resistor* enabled for this pin.

Since we now know that the GPIO 4 pin is *biased* to read back as *high*, we can wire up our button to make a connection with the opposite pole like this:

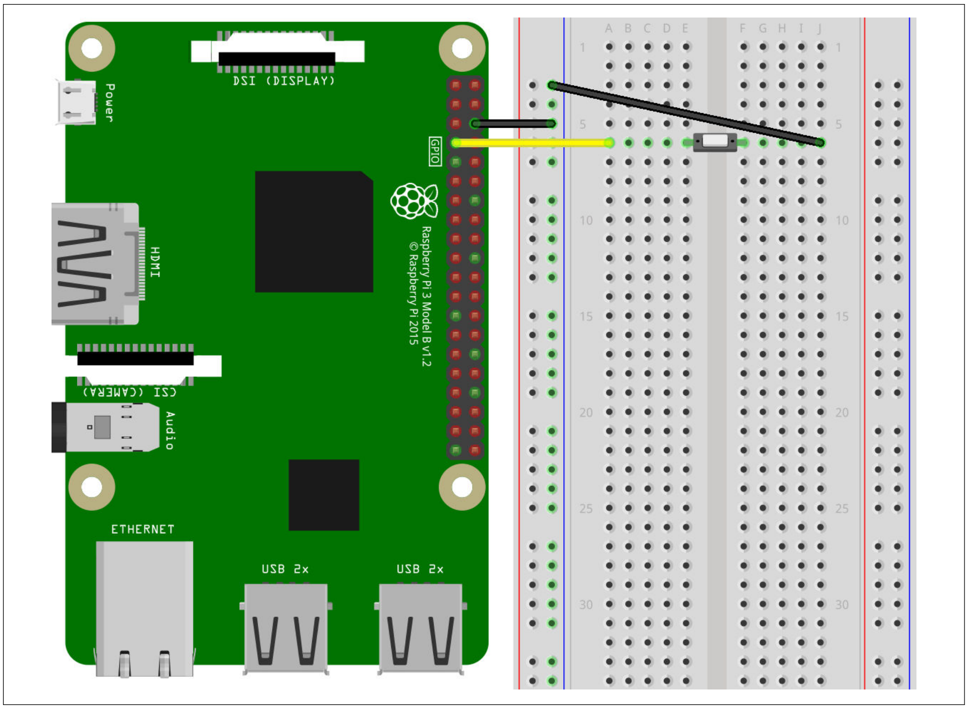

Figure 5-4. Wiring for Example 5-4

In one state the button, or switch, makes no connection to Ground. In this state, the pin will read back as it's default value, *high*. In the second state, there is a direct connection between Ground and the input pin, which is why the pin will read back as being *low*.

In the following sketch the window's background color is turned on and off by the connected button or switch:

Example 5-4.

```
import processing.io.*;

void setup() {
  GPIO.pinMode(4, GPIO.INPUT);
}

void draw() {
  if (GPIO.digitalRead(4) == GPIO.LOW) {
    // button is pressed, pin is low
    background(255, 255, 0);
  } else {
    // button is not pressed, pin is high
    background(0);
  }
}
```

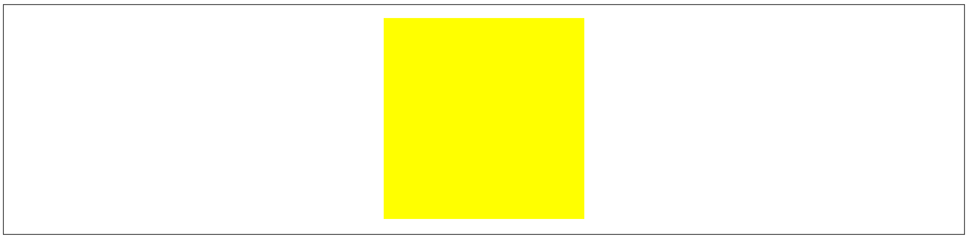

Figure 5-5. Display window after the button has been pressed

The `pinMode` function in combination with the `GPIO.INPUT` argument make the GPIO 4 pin act as an input. The `digitalRead` function returns either `GPIO.HIGH` or `GPIO.LOW`, the latter of which tints the background yellow.

When you select an input pin that is biased towards ground, such as is the case with GPIO 17 on the Raspberry Pi, your wiring would make a connection with 3.3V like so, and simply use the inverse logic of Example 5-4 in the sketch.

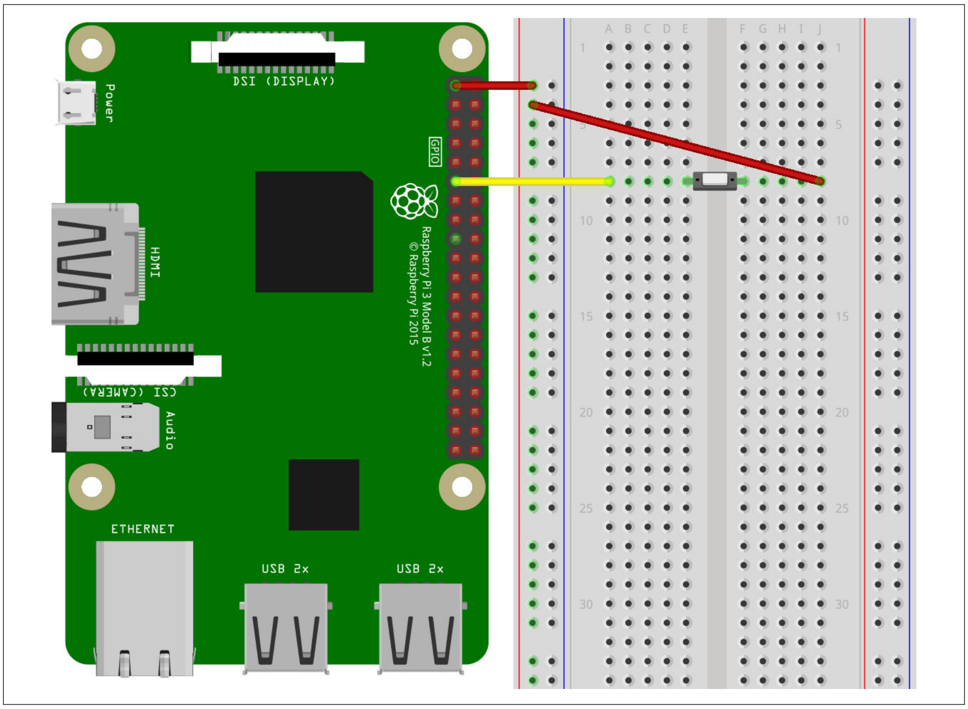

Figure 5-6. Wiring for Example 5-4 for a pin with pull-down resistor

Reacting to interrupts

The `attachInterrupt` function makes it so that the normal program flow gets interrupted whenever the value of an input pin changes, in which case a special function is being run.

There might be situations in a sketch should react instantly to external sensor readings, rather than merely the next time the `draw` function is being executed. Consider the case of a robot moving forward which has one or multiple micro-switches mounted to detect collisions. The breaks should be applied instantly when the robot encounters an obstacle, even if Processing is busy loading or saving files, or if we are in between drawing frames.

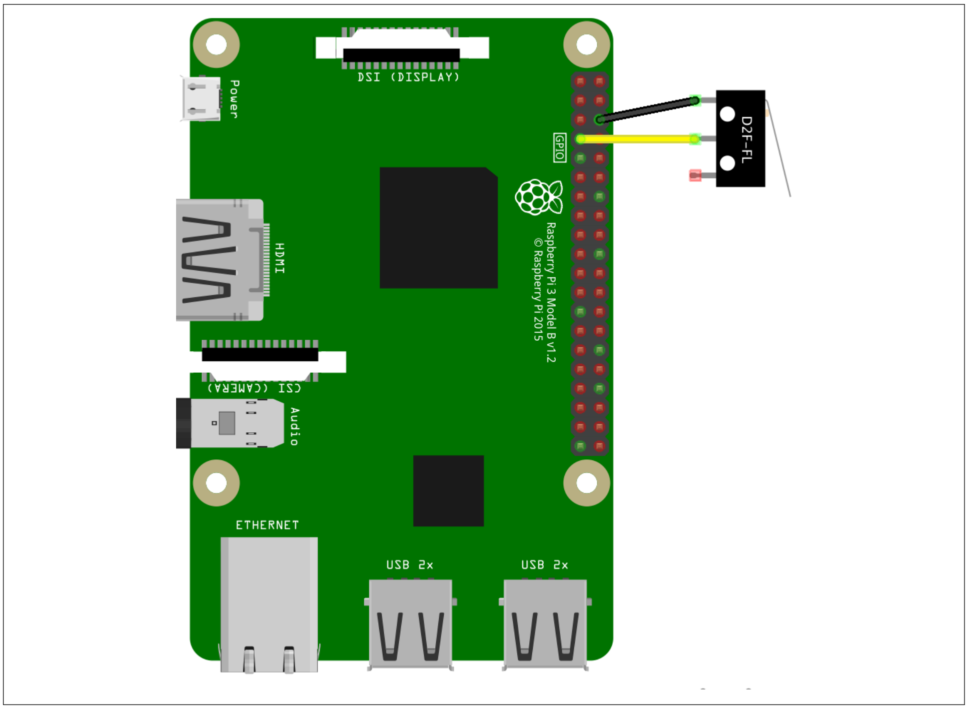

Figure 5-7. Wiring for Example 5-5

This sketch shows how the `attachInterrupt` function is being used to divert the normal program flow.

Example 5-5.

```
import processing.io.*;

void setup() {
  GPIO.pinMode(4, GPIO.INPUT);
  GPIO.attachInterrupt(4, this, "obstacle", GPIO.FALLING);
}

void draw() {
  // other functionality goes in here
}

void obstacle(int pin) {
  println("Hit obstacle on pin " + pin);
  // apply brakes
}
```

The first parameter to the `attachInterrupt` function is input pin to monitor. The second parameter is always `this` to refer to the current sketch. The third parameter is the name of the function to call, in this case the `obstacle` function. The last parameter determines for which events the function is going to be called. This needs to be one of the following:

GPIO.FALLING	value of pin switches from high to low
GPIO.RISING	value of pin switches from low to high
GPIO.CHANGE	value of pin changes in either direction

Notice how the `obstacle` function in the example above has a parameter `pin` of type `int`. This is helpful for using the same function for interrupts from multiple pins. Instead of a single micro-switch our robot might, for example, have one on the left front and one on the right front, and we would could use the parameter to the interrupt function to determine which way to go.

Example 5-6.

```
void obstacle(int pin) {
  println("Hit obstacle on pin " + pin);
  if (pin == 4) {
    // left sensor, try going right
  } else {
    // right sensor, go left
  }
}
```

The `releaseInterrupt` function allows to stop listening for interrupt again. Its single parameter is the number of the input pin.

Example 5-7.

```
void obstacle(int pin) {
  println("Hit obstacle on pin " + pin);
  GPIO.releaseInterrupt(pin);
  // now we won't get future notifications from this pin
}
```

The `noInterrupts` and `interrupts` function meanwhile allows to *temporarily* disable handling of interrupts. This is useful when interrupts from other pins could happen at the same time, yet our handling of the first interrupt shouldn't be interrupted by a different one.

Example 5-8.

```
void obstacle(int pin) {
  noInterrupts();
  // this code will run uninterrupted
  println("Hit obstacle on pin " + pin);
  // apply brakes
  // enable interrupts again
  interrupts();
}
```

Interrupts on the *same pin* are automatically prevented from occuring while the program handles an earlier one.

Releasing the pin

Closing the display window, or terminating the sketch via the *Stop* button, will make the pins remain in whatever state they are at that point. A pin configured as output and set to light up an LED will continue to do so after the sketch has been closed, which is at times useful.

To instead undo the change made by the sketch, and to hand the pin back to the operating system, call the `releasePin` function. This might be neccessary for re-use a GPIO pin with certain other programs outside of Processing.

Example 5-9.

```
import processing.io.*;

void setup() {
  GPIO.pinMode(4, GPIO.OUTPUT);
  GPIO.digitalWrite(4, GPIO.HIGH);
}

void draw() {
  // other functionality goes in here
}

void mousePressed() {
  // clicking the mouse releases the pin and closes the sketch
  GPIO.releasePin(4);
  exit();
}
```

I²C

I²C is a communication bus that allows the SBC to exchange data with other integrated circuits. It consists of a "master" device and one or multiple "slave" devices. In the context of Processing's Hardware I/O library, the SBC is always takes the role of the "master" device.

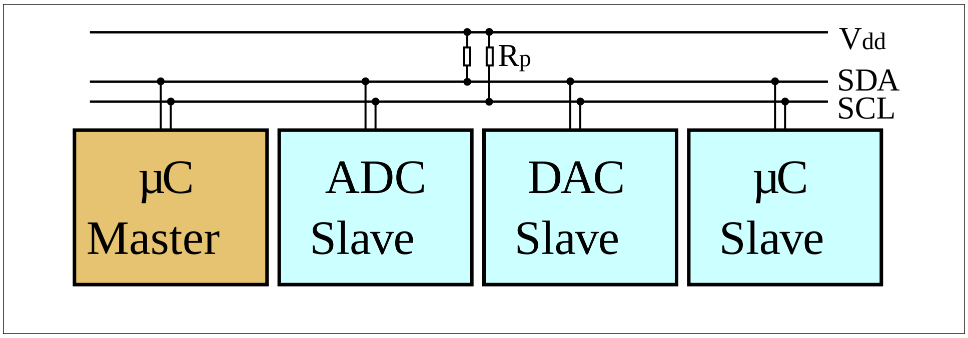

Figure 5-8. I²C schematic

Two lines connect each device in the bus: one for data, called *SDA* (*Serial Data Line*), and one for a clock signal generated by the "master" device, called *SCL* (*Serial Clock Line*). SBCs such as the Rapsberry Pi have special pins on their expansion headers where those lines are to be connected.

Some schematics might suggest it to be necessary to connect *pull-up* resistors between *SDA* and *3.3V*, as well as between *SCL* and *3.3V*. This is superfluous as the Raspberry Pi and other SBCs already have internal *pull-ups* for those lines.

The communication over I²C is always initiated by the "master" device, which only talks to a single peripheral at a time. Each "slave" device comes with a fixed 7-bit address, which can be found in the respective datasheet documents. (Some devices also allow the address to be altered by a jumper. This way more than one device of a given type can be part of the same bus.) The "master" device will transmit the address of the device it wants to talk to, and will then either continue by sending a number of bytes to the device, or requesting to receive bytes from it. Afterwards, the bus is instantly ready for a future exchange, which will function following the same pattern.

The Hardware I/O library expects the addresses to be given in 7-bit form, same as for *Arduino's Wire* library or the output of the `i2cdetect` utility. However, some devices' datasheet have the address in the higher 7 bites of the address, with the lowest bit determining whether the device is to be written to (0), or read from (1).

If you encounter an address greater than 119, or, a datasheet lists two addresses for a device - one for read and one for write - that differ exactly by one, this is an indication that this is the case. To use this address with Processing you need to shift its value one bit to the right.

Selecting the I²C interface

The following sketch prints all I²C interfaces the IO library can work with:

Example 5-10.

```
import processing.io.*;
printArray(I2C.list());
```

On the Raspberry Pi, the `i2c-2` interface is used by the operating system to identify so-called HATs (expansion boards), and thus shouldn't be used. If you are missing the `i2c-1` interface Processing can use, make sure it is enabled in the *Raspberry Pi Configuration* utility.

Writing data over I²C

The following example will use a MCP4725 Digital-to-Analog converter from Microchip. Digital-to-Analog converters, or *DAC*, convert digital signals into analog voltages. Those analog voltages can then, for example, be used to dim LEDs, or to generate audio signals.

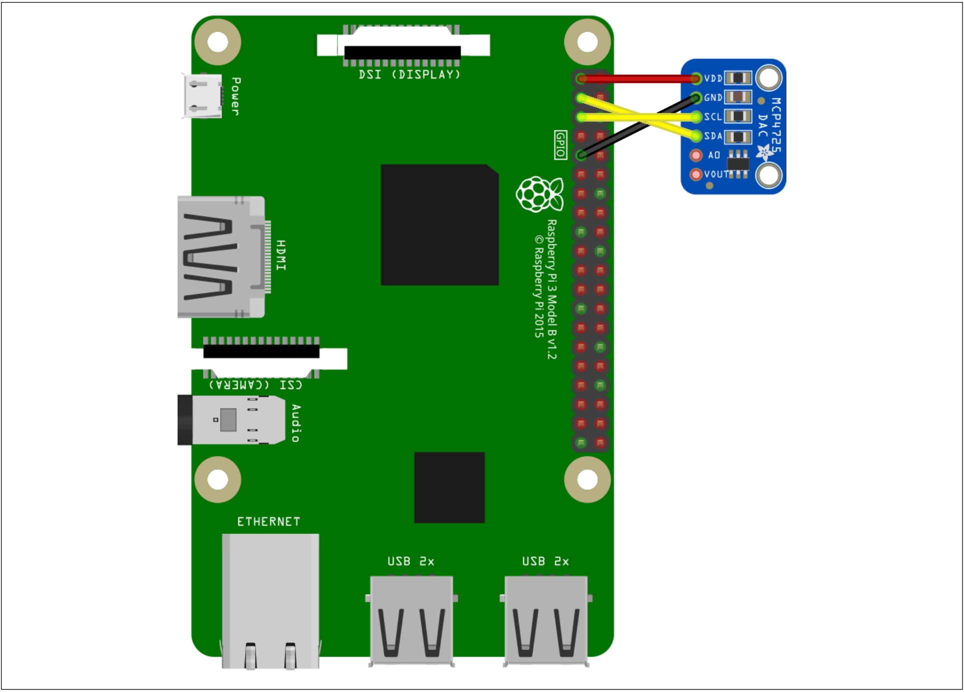

Figure 5-9. Wiring for Example 5-11

Example 5-11. Writing data over I²C

```
import processing.io.*;
I2C i2c;

void setup() {
  // you might need to use a different interface on other SBCs
  i2c = new I2C("i2c-1");
}

void draw() {
  background(255);
  line(mouseX, 0, mouseX, 99);

  int val = int(4095 * map(mouseX, 0, 99, 0.0, 1.0));
  i2c.beginTransmission(0x60);
  i2c.write(val >> 8);
  i2c.write(val & 255);
  i2c.endTransmission();
}
```

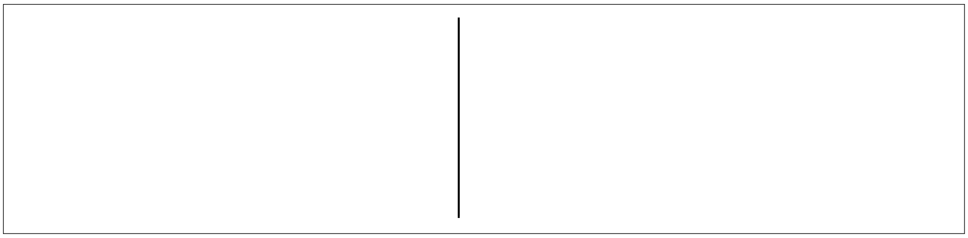

Figure 5-10. Display window with the DAC output being about 1/3 of 3.3V

The sketch varies the voltage output by the *DAC* depending on the mouse position. The line `i2c = new I2C("i2c-1");` creates a new I2C object that uses the `i2c-1` interface. In the `draw` function, we use Processing's `map` function to convert the X coordinate of the mouse pointer into the range of zero to one. The MCP4725 expects its value to be given as a 12-bit number. For this reason we multiply by 4095. The `beginTransmission` method has as its parameter the device address, which is here hexadecimal 60. The `write` method is used to transfer one byte of the value at a time, in the way that is described by the MCP4725's datasheet. The `endTransmission` method then actually sends the communication on its way.

If the communication is not successful, this will throw an exception and print the message to the Console: `The device did not respond. Check the cabling and whether you are using the correct address.`

For troubleshooting I²C issues the `i2cdetect` utility can be used in the Terminal. The command **`i2cdetect -y 1`** will probe `i2c-1` and list all addresses that responsed on the bus. This is helpful for finding the correct address of devices that do not respond in Processing. Here is how to install `i2cdetect` if it is not already installed on your SBC: `sudo apt-get install i2c-tools`

Reading data over I²C

The following example will use a HMC6352 compass module from Honeywell to read in the measured heading. (There are more advanced parts available, such as the MPU-9150 from InvenSense, but we will use the HMC6352 here for its ease of use.)

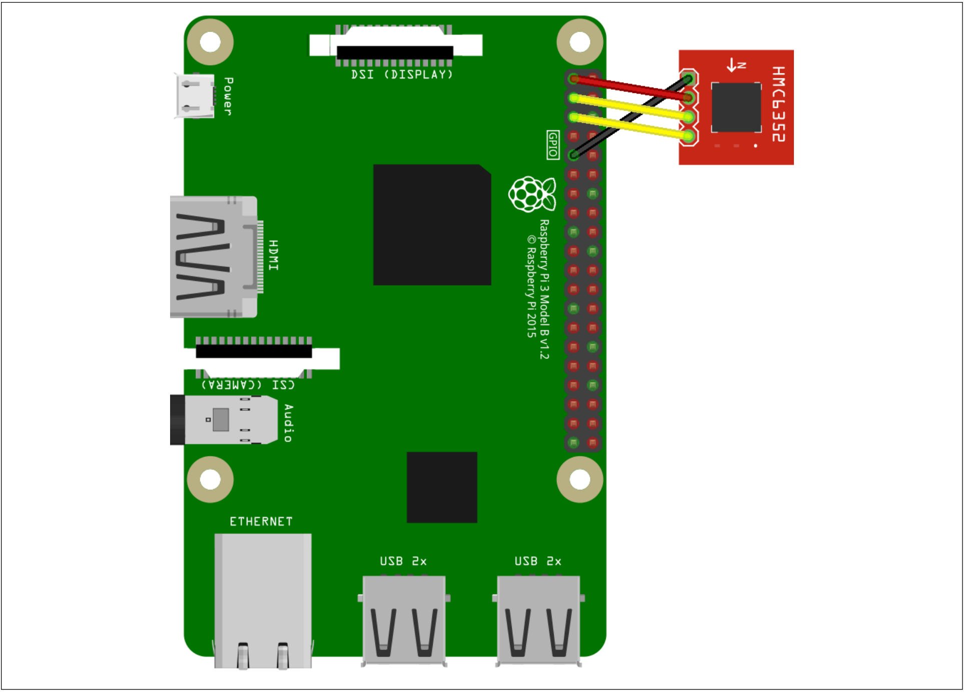

Figure 5-11. Wiring for Example 5-12

Example 5-12.

```
import processing.io.*;
I2C i2c;

void setup() {
  // you might need to use a different interface on other SBCs
  i2c = new I2C("i2c-1");
}

void draw() {
  background(255);
  float deg = getHeading();
  println(deg + " degrees");
  line(50, 50, 50 + sin(radians(deg))*50, 50 - cos(radians(deg))*50);
}

float getHeading() {
  i2c.beginTransmission(0x21);
  i2c.write(0x41);
  byte[] in = i2c.read(2);
  i2c.endTransmission();
  // put bytes together to tenth of degrees
  int deg = (in[0] & 0xff) << 8 | (in[1] & 0xff);
  // return degrees
  return deg / 10.0;
}
```

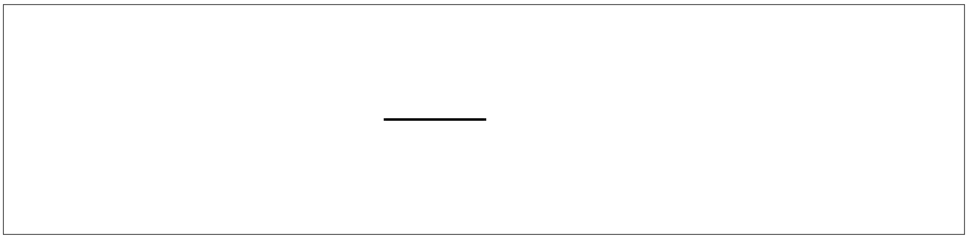

Figure 5-12. Display window heading west

The `beginTransmission` method has as its parameter the device's address, which is hexadecimal 21, according to the datasheet of the HMC6352. The following protocol for retrieving the heading is also described in the datasheet, as it very much varies from product to product and manufacturer to manufacturer: the byte `0x41` commands the module to prepare the heading data, which is then read with the `read` method, and stored into an array of bytes (`byte[]`). The single parameter to `read` holds the number of bytes we're expecting, which is two. The `endTransmission` method finishes the exchange.

As the heading is returned in two bytes, bit-shifting (`<<`) and a bitwise OR (`|`) is used to re-combine it into a single variable. You might wonder about the `& 0xff` (bitwise AND with 255). This is to interpret the `byte` variable, which in Processing has a range from -128 to 127, into an `int` variable with range 0 to 255. This is a helpful pattern when interfacing with electronics, where one is most typically working with unsigned data types (such as the `byte` in *Arduino*).

There can only be a single `read` as part of a transmission, which also needs to always come last. Any number of `write` operations can preceed the `read`, such as the single one in this example. Those will be batched and sent together with the `read`.

Writing Object-oriented code

The example Example 5-11 can be also written in an object-oriented way like so: (In Processing it is customary to put classes into their own tab, which done by clicking the arrow button in the right-most tab and selecting *New Tab*. Give the file the same name as your class, like "MCP4725" for our example.)

Example 5-13. MCP4725.pde

```
import processing.io.I2C;

class MCP4725 extends I2C {
  int address;

  MCP4725(String dev, int address) {
    super(dev);
    this.address = address;
  }

  void setAnalog(float fac) {
    int val = int(4095 * constrain(fac, 0.0, 1.0));
    beginTransmission(address);
    write(val >> 8);
    write(val & 255);
    endTransmission();
  }
}
```

The `MCP4725` class inherits from the generic `I2C` class and implements a specialized method, `setAnalog`. Here is how to use this class in the main sketch file (always the left-most tab):

Example 5-14. Main sketch

```
import processing.io.*;
MCP4725 dac;

void setup() {
  // you might need to use a different interface on other SBCs
  dac = new MCP4725("i2c-1", 0x60);
}

void draw() {
  background(255);
  line(mouseX, 0, mouseX, 99);
  dac.setAnalog(map(mouseX, 0, 99, 0.0, 1.0));
}
```

This technique is very useful when interfacing with multiple I^{2}C devices in one sketch.

In case of the MCP4725, we can add a second *DAC* to the bus by connecting a specific pin by one of the two to *3.3V*. This will make this one use address `0x61`, instead of `0x60`, that the first one uses. To drive two devices, we modify Example 5-14 like so:

Example 5-15.

```
MCP4725 dac1, dac2;

void setup() {
  // you might need to use a different interface on other SBCs
  dac1 = new MCP4725("i2c-1", 0x60);
  dac2 = new MCP4725("i2c-1", 0x61);
}

void draw() {
  background(255);
  line(mouseX, 0, mouseX, 99);
  dac1.setAnalog(map(mouseX, 0, 99, 0.0, 1.0));
  // make the second one work in the opposite manner
  dac2.setAnalog(map(mouseX, 0, 99, 1.0, 0.0));
}
```

Having different I^2C devices in the same sketch, that each are implemented as a class and inherit from *I2C*, would work analogously. With the devices used in this chapter, one could, e.g., easily read the heading from a `HMC6352` device, and use it to set the voltage one one or more `MCP4725` parts.

SPI

SPI (*Serial Peripheral Interface*) is another widely-used communication bus that the SBC can use to exchange data with other integrated circuits. Like I^2C, it consists of a "master" device and one and more "slave" devices, whereas the SBC will take the role of the "master" device in the context of Processing's Hardware I/O library.

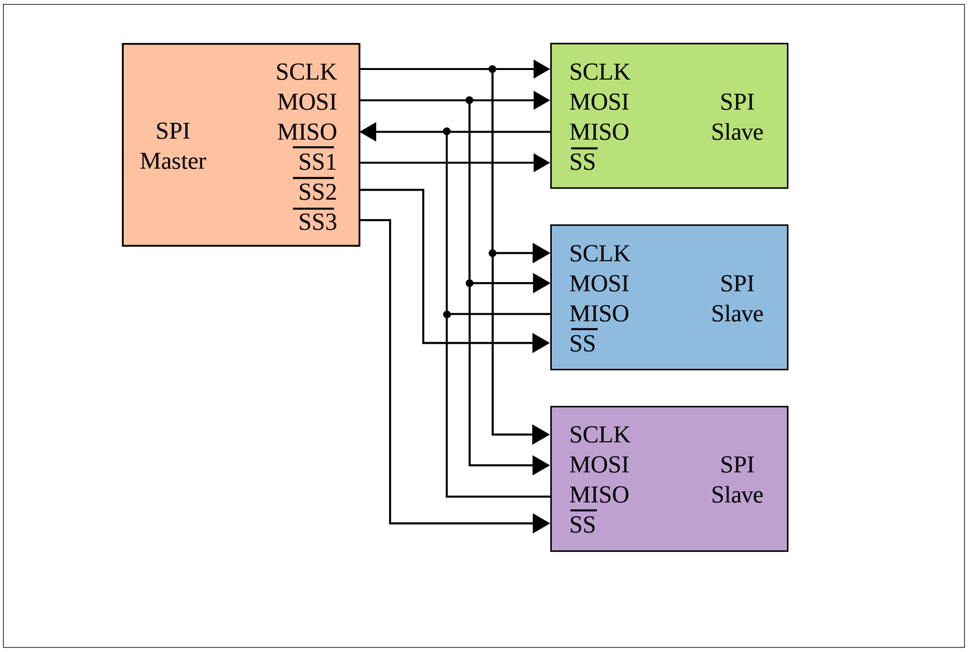

Figure 5-13. SPI schematic

Four lines are used with SPI: a clock signal generated by the master (*SCLK*), data being transmitted *to* the “slave” device (*MOSI*, for *Master Output, Slave Input*), data being transmitted *from* the “slave” device (*MISO*, for *Master Input, Slave Output*), as well as one line per peripheral to select which one to talk to (*SS*, for *Slave Select*).

SPI is a synchronous interface in the sense that in every transmission there is an equal number of bytes sent (over *MOSI*) as well as received (on *MISO*). One is not possible without the other. A transmission starts by the the “master” pulling the *SS* line of one peripheral low (to *GND*). A number of bytes is being exchanged. The peripheral is then then automatically de-selected by its *SS* line being raised to default high (*3.3V*) level.

Selecting the SPI interface

The following sketch prints all SPI interfaces the IO library can work with:

Example 5-16.

```
import processing.io.*;
printArray(SPI.list());
```

On the Raspberry Pi, there are two interfaces, `spidev0.0` and `spidev0.1`. (If you don’t see either, you might need to enable the SPI interface in the *Raspberry Pi Config-*

uration utility.) Both interfaces share the majority of their pins, except `spidev0.0` will pull GPIO pin 8 on the header (labeled *CE0*) low, whereas `spidev0.1` will do the same with GPIO pin 7 (labeled *CE1*). This way, two SPI peripherals can be interfaced with, simply by alternating between the interfaces.

To use more than two SPI peripherals, use regular GPIO pins instead of the SBC's dedicated *SS* pins, and control their voltages with the `digitalWrite` function (see XXX).

Transmitting data over SPI

The following example will use a MCP3001 Analog-to-Digital converter from Microchip. Analog-to-Digital converters, or *ADC*, convert analog voltages to digital values.

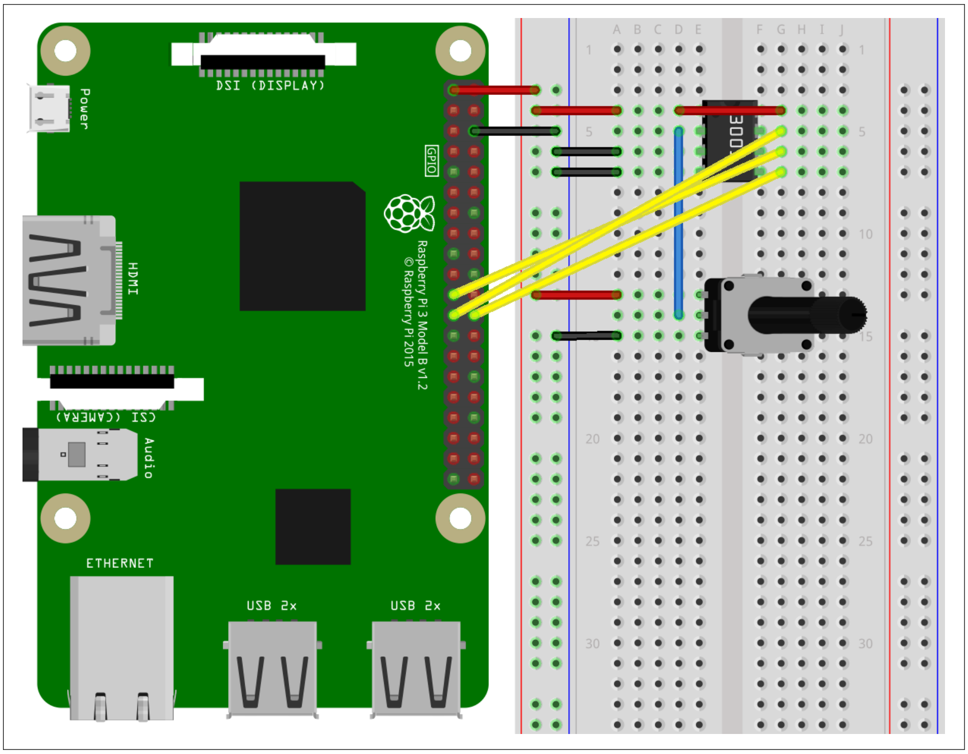

Figure 5-14. Wiring for Example 5-17

Example 5-17.

```
import processing.io.*;
SPI spi;
```

```
void setup() {
  // you might need to use a different interface on other SBCs
  spi = new SPI("spidev0.0");
}

void draw() {
  byte[] out = { 0, 0 };
  byte[] in = spi.transfer(out);
  int val = ((in[0] & 0x1f) << 5) | ((in[1] & 0xf8) >> 3);
  // val is between 0 and 1023
  background(map(val, 0, 1023, 0, 255));
}
```

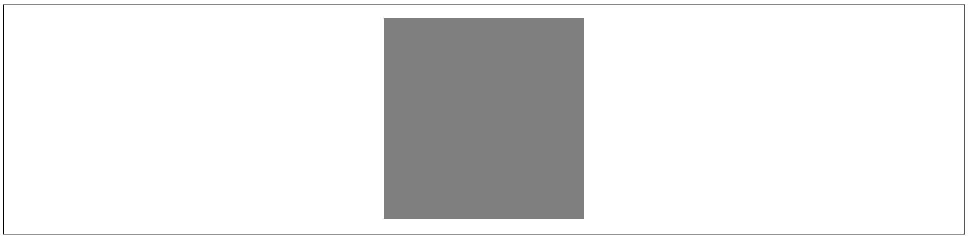
Figure 5-15. Display window with potentiometer half-way

The line `spi = new SPI("spidev0.0");` creates a new SPI object that uses the `spidev0.0` interface. The array of bytes `out` contains two bytes that are sent to the *ADC*. For the `MCP3001` the content of those bytes isn't relevant, there is not even a pin for *MOSI*, but we ought to send out some data, due to the synchronous nature of the interface. The `transfer` function does the actual exchange, and writes the value it receives into the array of bytes `in`.

Unlike with I^{2}C, there is no way of checking whether the transmission was sucessful built into the protocol. The `transfer` method will return two bytes also if the `MCP3001` isn't even connected or powered. Keep this in mind when troubleshooting, and check the values received for plausibility if you can.

The SPI bus can be operated in different modes, which can be set with the `settings` method. The default settings the same as with Arduino's *SPI* library (*500 kHz, most-significant bit first, SPI MODE0*), and should work with most peripherals. See the library's reference for other options.

Writing Object-oriented code

The example Example 5-17 can be also written in an object-oriented way like so:

Example 5-18. MCP3001.pde

```
import processing.io.SPI;

class MCP3001 extends SPI {

  MCP3001(String dev) {
    super(dev);
  }

  float getAnalog() {
    byte[] out = { 0, 0 };
    byte[] in = transfer(out);
    int val = ((in[0] & 0x1f) << 5) | ((in[1] & 0xf8) >> 3);
    // val is between 0 and 1023
    return val/1023.0;
  }
}
```

Here we use two *MCP3001* objects, each connected to a different interface, to draw lines on the screen:

Example 5-19. Main sketch

```
MCP3001 adc1, adc2;

void setup() {
  size(300, 100);
  // you might need to use different interfaces on other SBCs
  adc1 = new MCP3001("spidev0.0");
  adc2 = new MCP3001("spidev0.1");
}

void draw() {
  background(255);
  float val1 = adc1.getAnalog();
  float val2 = adc2.getAnalog();
  line(val1 * width, 0, val1 * width, height);
  line(val2 * width, 0, val2 * width, height);
}
```

The `width` and `height` variables in Processing always contain the width and height of the display window, as defined in the `size` function above.

Analog pins

Measuring analog voltages works with the `analogRead` function on Arduino, which uses the microcontroller's built-in Analog-to-Digital converter. The current generation of SBCs commonly lacks this ability, but the following section will discuss three strategies how analog sensors can be interfaced with after all.

Using an external Analog-to-Digital converter

See Example 5-17 for how to use an inexpensive *ADC* connected via *SPI*. This option gives 10 bits of resolution, which makes the precision of the measurement comparable with Arduino's analog pins. To also match the number of analog pins available on Arduino, the related part *MCP3008* can be used, which has eight different input channels that can be measured from. See the *SPIAnalogDigitalOOP8* example that comes with the Hardware I/O library for how to use the *MCP3008* with Processing.

Using a digital pin to measure two states

A photocell might inform us about various shades of lightness, yet in certain applications it is only being used to determine *whether or not* a person is walking by it. In other words: it is just used to probe whether or not a certain threshold of lightness (or shadow) is being met. In the diagram below, the setting on a trim potentiometer will determine this threshold, and whether a digital input pin will read as high or low for a given light intensity. (SBCs will typically switch between interpreting a voltage as high or low at about half-way between *0V* and *3.3V*.)

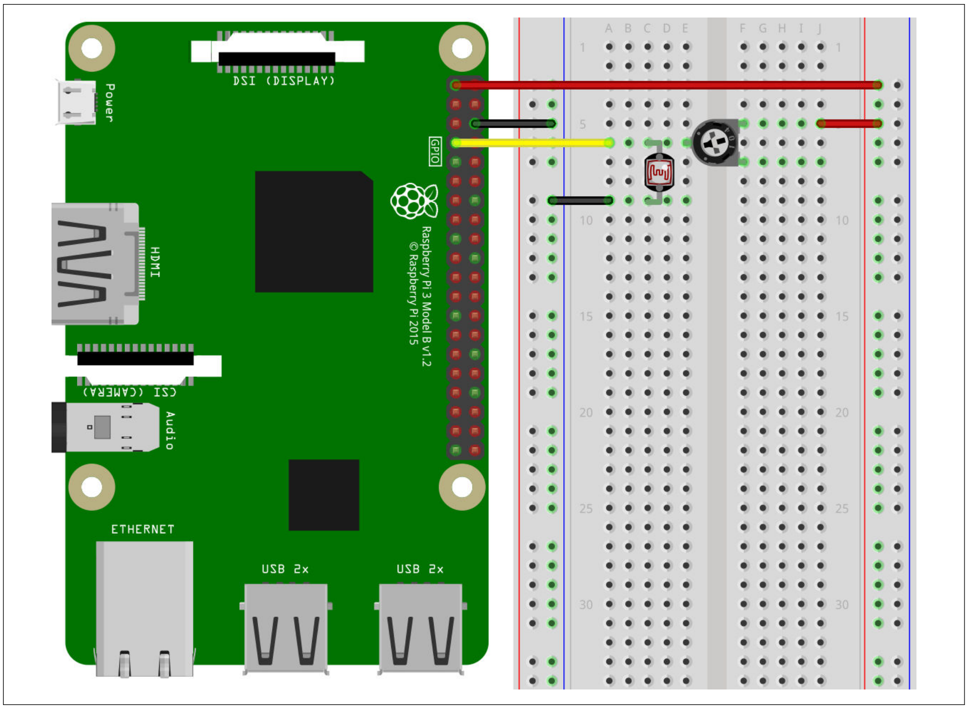

Figure 5-16. Wiring diagram

Using a digital pin in combination with a capacitor

Electrical engineer Limor Fried popularized this method of getting an approximate reading from an analog resistive sensor, by using just a digital pin and a capacitor: the capacitor gets discharged by setting the pin to output and low. Once discharged, the time it takes to charge the capacitor, until the pin - switched to input - reads as high, is being measured. The higher the resistance of the sensor, the more time will pass for the capacitor to reach this level, and vice versa. See the *SimpleResistorSensor* example that comes with the Hardware I/O library for details.

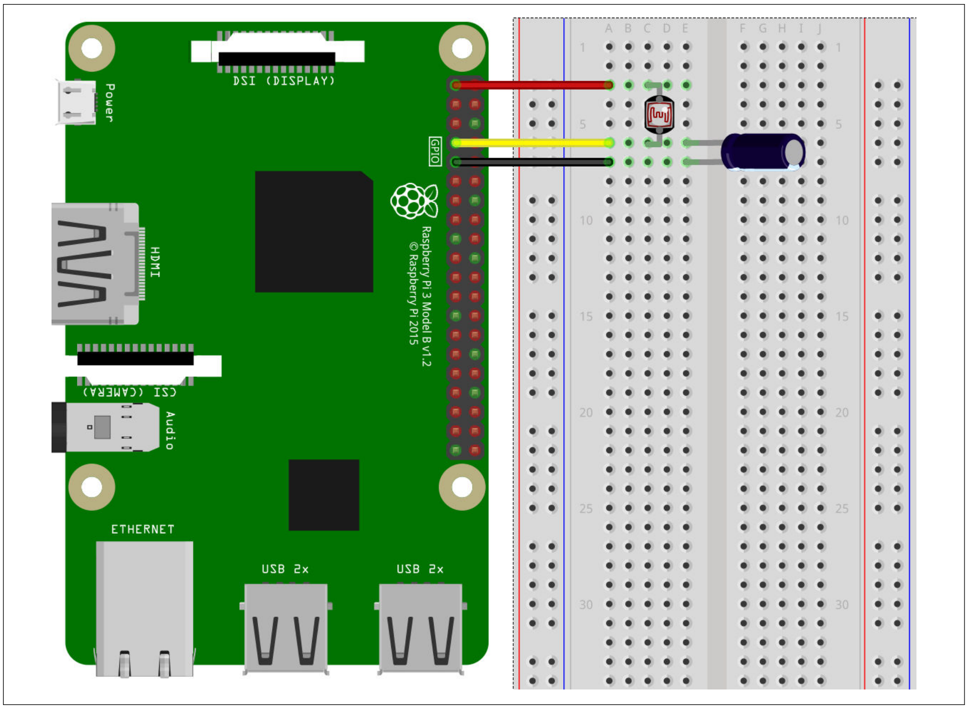

Figure 5-17. Wiring diagram

Servo motors

Concentional RC servo motors are a quick way to translate many different types of dynamics and motions from inside your sketch to the world outside. Objects connected to mounting brackets can be made to rotate, cable can be spooled and unspooled, and systems out of pulleys and levers can have impressive outcomes.

Servo motors can be used with any digital pin, by using the `SoftwareServo` class like so:

Example 5-20.

```
import processing.io.*;

SoftwareServo servo;

void setup() {
  size(300, 100);
  servo = new SoftwareServo(this);
  servo.attach(4);
}
```

```
void draw() {
  background(255);
  float angle = 90 + sin(frameCount/100.0) * 85;
  servo.write(angle);
  float x = map(angle, 0, 180, 0, width);
  line(x, 0, x, height);
}
```

The `attach` method connects `SoftwareServo` object to GPIO pin 4, whereas the `write` method expects an angle from 0 to 180 degrees. In the example, we're using Processing's `frameCount` variable, which is always incremented each time through `draw`, as the parameter to the sine function. The result is multiplied only by 85 (instead of the expected 90), as to not drive the servo exactly to its limits, which could cause a stall and high current consumption.

To momentarily pause a servo motor, use the `detach` method. Our example can be extended like this in order to detach, and re-attach, the servo upon mouse click.

Example 5-21.

```
void mousePressed() {
  if (servo.attached()) {
    servo.detach();
  } else {
    servo.attach(4);
  }
}
```

While the Raspberry Pi has a pin for 5V that could be sufficent for smaller servo motors without load, the diagram below shows how to power one or more motors with an external power supply. (Common servos expect a supply voltage of ca. 4.8 to 6 Volts.)

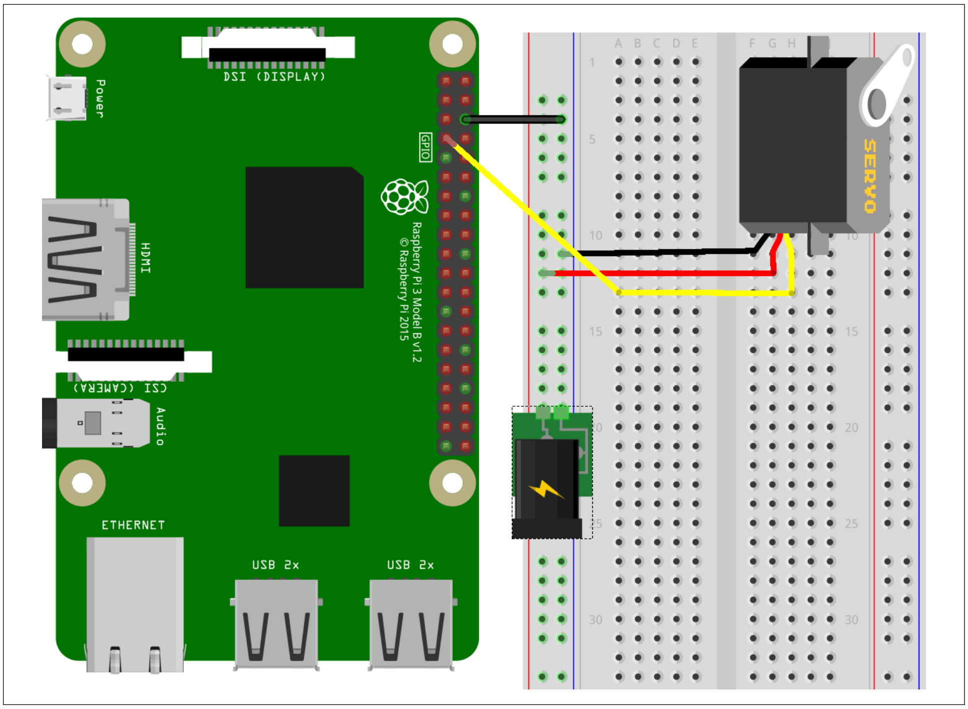

Figure 5-18. Wiring for Example 5-20

PWM

To make use of PWM currently requires Processing to run with the privileges of the `root` user. Support for this technique is not very mature at this point. Here be dragons!

Pulse-width modulation, or *PWM*, is a technique for approximating an analog signal by repeatedly turning a digital output on and off at a high frequency. The greater the proportion of the time the output is *on*, relative to the time it is *off*, the higher the analog value. This is called the "duty cycle".

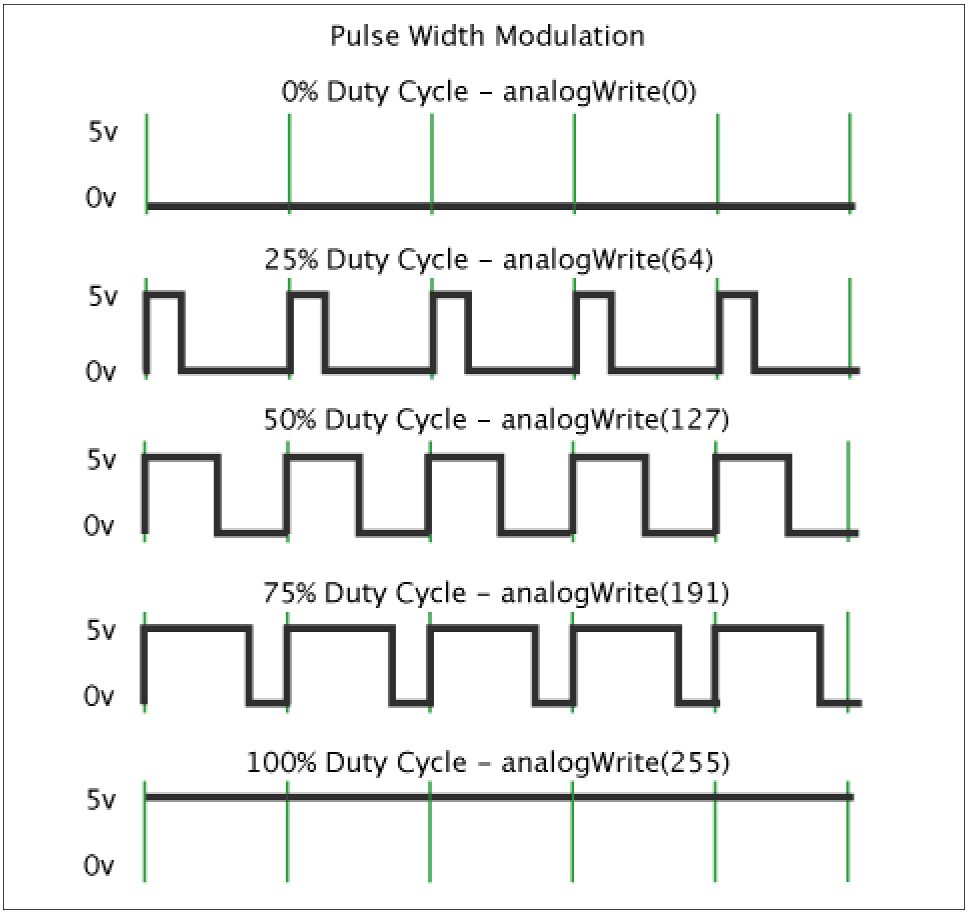

Figure 5-19. PWM duty cycle

Arduino's `digitalWrite` function makes use of *PWM* and is limited to certain pins that support this functionality. The same is also true for SBCs. The following sketch lists all available *PWM* channels:

Example 5-22.

```
import processing.io.*;
printArray(PWM.list());
```

On the Raspberry Pi, the line `dtoverlay=pwm-2chan` needs to be added to the `/boot/config.txt` file, and the board rebooted, before the following channels are accessible:

- `pwmchip0/pwm0`, which is connected to GPIO pin 18, and

- `pwmchip0/pwm1`, connected to GPIO pin 19

Both channels appear to be shared with the Pi's analog audio output, so only actively use one or the other.

The following sketch makes use of a *PWM* channel in Processing:

Example 5-23.

```
import processing.io.*;
PWM pwm;

void setup() {
  size(300, 100);
  // you might need to use a different channel on other SBCs
  pwm = new PWM("pwmchip0/pwm0");
}

void draw() {
  background(255);
  pwm.set(map(mouseX, 0, width, 0.0, 1.0));
  line(mouseX, 0, mouseX, height);
}
```

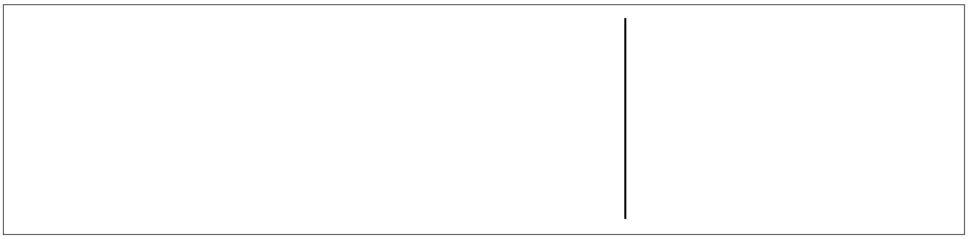

Figure 5-20. Display window with duty cycle at about 75%

The `set` method updates the channel's duty cycle from 0.0 (0%) to 1.0 (100%), with a frequency of 1 kHz. A different frequency can be specified as the second parameter to `set`. This frequency is in Hz.

Use the `clear` method to disable the modulation. The following addtion makes the sketch also exit when the *Escape* key is being pressed:

Example 5-24.

```
void keyPressed() {
  if (key == ESC) {
    pwm.clear();
    exit();
  }
}
```

Waiting

Working with physical hardware devices quickly leads to situation where the program is out to *wait* for a certain amount of time to pass, or an event to occur. The datasheet of a device might instruct you to wait a certain number of milliseconds after waking it up from sleep, before commanding it to do a certain action. Or you might have determined that a servo motor should spool up two seconds worth of cable. Like *Arduino*, Processing has a `delay` function, which is used to wait for a certain number of milliseconds (thousands of a second) like so:

Example 5-25.

```
delay(10);
```

This is fine for shorter durations, but with longer time spans the user will notice that the screen is no longer updating and responding to input. This is because the `delay` function *blocks* the execution of the sketch until the set time has expired. To integrate delays into a sketch in a way that they don't interfere with the normal program flow, make use of `millis`. This function returns the number of milliseconds elapsed since the program has started. The following example will make use of `millis` to spool and unspool cable with a continuous rotation servo motor, while drawing to the screen and remaining responsive to user input.

Example 5-26.

```
import processing.io.*;
SoftwareServo servo;
int start = 0;

void setup() {
  servo = new SoftwareServo(this);
  servo.attach(4);
}

void draw() {
  int now = millis();

  if (now < 2000) {
    // spool
    servo.write(180.0);
  } else if (now-start < 4000) {
    // remain stationary (servo in neutral)
    servo.write(90.0);
  } else if (now-start < 6000) {
    // unspool
    servo.write(0.0);
  } else {
    // end of the sequence, reset
    start = now;
  }
}
```

Often times a sketch will also wait for an input pin to to change its value, either to be *high* (a process called *raising*), or to *low* (a process called *falling*). The IO library has a function to help with this, `waitFor`. The following line will pause until GPIO pin 4 is *low*. The pin needs to be set to input using the `pinMode` function ahead of time.

Example 5-27.

```
GPIO.waitFor(4, GPIO.FALLING);
```

If we wanted to make sure the pin does not take longer than 10 milliseconds to to become *low*, we can specify this timeout as the third parameter to this function.

Example 5-28.

```
GPIO.waitFor(4, GPIO.FALLING, 10);
```

If the pin does not behave as expected, for example, because a peripheral is not wired up correctly, this will print the exception message "Timeout occurred" in Processing Desktop Environment's Message Area, and highlight the line that caused it. This is tremendously helpful with multiple places in the sketch that could have caused execution to get "stuck".

Your sketch can also handle those exceptions itself by wrapping one or more `waitFor` calls in a try-catch block:

Example 5-29.

```
try {
  GPIO.waitFor(4, GPIO.FALLING, 10);
  // ...
  GPIO.waitFor(17, GPIO.RISING, 10);
} catch (RuntimeException e) {
  println("Error initializing hardware");
}
```

Limitations of Hardware I/O

How do the IO capabilities of a SBC, such as the Raspberry Pi 3, compare to those of an 8-bit microcontroller, such as the venerable *Arduino Uno*? A fundamental distinction becomes obvious when we look at the overall system architectures, and consider where our sketch will be situated.

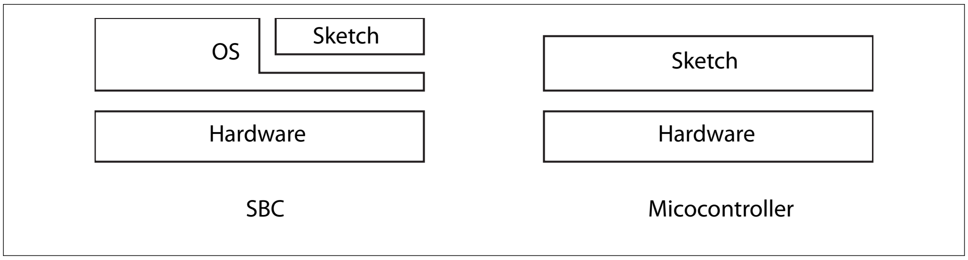

Figure 5-21. Architecture comparison

The code written in *Arduino* is being directly executed by the microcontroller's CPU, without any operating system or *background tasks* potentially in the way. This leads to a very predictable, very repeatable execution, more so since those microcontrollers aren't necessarily networked. A sketch in Processing, by comparison, sits on top of a feature-rich *multitasking* operating system, which allows many different programs and tasks to be executed simultaneously, in a fashion that quickly alternates between them. Since the flavor of Linux used on SBCs is generally a general-purpose type, aimed at desktop and server-type machines, this can lead to a bit of jitter. The indirection of accessing the hardware only through the operating system is also quite a bit slower.

What does this mean in practice? Flipping the value of a digital output pin in Processing on a Raspberry Pi 3 with default configuration currently takes about 0.21 ms (σ = 0.05 ms, n = 400), whereas the same operation on an *Arduino Uno* takes a mere 3.45 µs (σ = 0.35 µs, n = 400). Sleeping 10 milliseconds using the `delay` function before changing a digital output pin value has a standard deviation of 0.14 ms with Processing on the Raspberry Pi 3 (n = 200), where on the microcontroller it is only 1.79 µs (n = 200).

While some of this deficit could be overcome by careful configuration, e.g. by using a specialized kernel optimized for this type of application, the Processing project believes that portability, ease of use, and the integration into a mainstream Desktop operating system are more worthwhile goals to pursue. The project is committed to building tools that work well in a wide area of applications, but if your application is requiring more performace or narrower tolerances than what this "software sketchbook" offers, you might need to use specialized tools that are "closer to the metal", meaning the hardware.