New Devices Lab exercise 1: Arduino Programming

Pieter Koopman

5 February 2018

1 Introduction

Small computers drive many new devices and elements in the Internet of Things, IoT. Such a computer is often a System on a Chip, SoC, a single IC that integrates all components of a computer on a single chip. Otherwise the microcomputer is usually a single printed circuit board that contains all parts. The archetype of such systems is the Arduino Uno R3. It is very widely used to construct new devices. Nowadays there are many variants of this Arduino and many similar systems available.

Arduino started as a student project in 2005. Currently there are two competing organizations developing and distributing the open source hardware and associated software. There is the original Arduino organization at `arduino.cc`, and the Arduino foundation at `arduino.org`.

1.1 Arduino Uno Hardware

The Arduino Uno in release 3, R3, is powered by an 8-bit ATmega328 microprocessor running at 16 MHz. It provides 32 KB of flash Memory. This memory is used to store programs. The contents of flash memory remain unchanged without power. When you power up an Arduino it contains the program last stored and starts executing it. The boot-loader is a small piece of code that helps to store a program in this flash memory. This boot-loader consumes 0.5 KB of the available flash memory.

The ATmega328 microprocessor offers 2 KB of RAM memory. The content of this memory is erased without power. This part of the memory is used to store the variables and contains the stack used to implement function calls.

These specifications show are similar to ordinary computers of 30 years ago. The popularity of the Arduino is due to its 14 digital I/O pins, of which 6 provide Pulse Width Modulation (PWM) output, and 6 analog I/O pins. These pins are named `D0..D13` and `A0..A5`. The analog pins contain a 10-bit Analog to Digital Converter, ADC, and the associated Digital to Analog Converter, DAC. These converters map analog signals between 0 and 5 volts to numbers in the range from 0 to 1023, and vice versa. The program controls whether pins are used as input or output. These I/O pins are very convenient to read or control various kind of input output peripherals. The digital pins are at the topside of the board in Figure 1, and the analog Pins are at the bottom right.

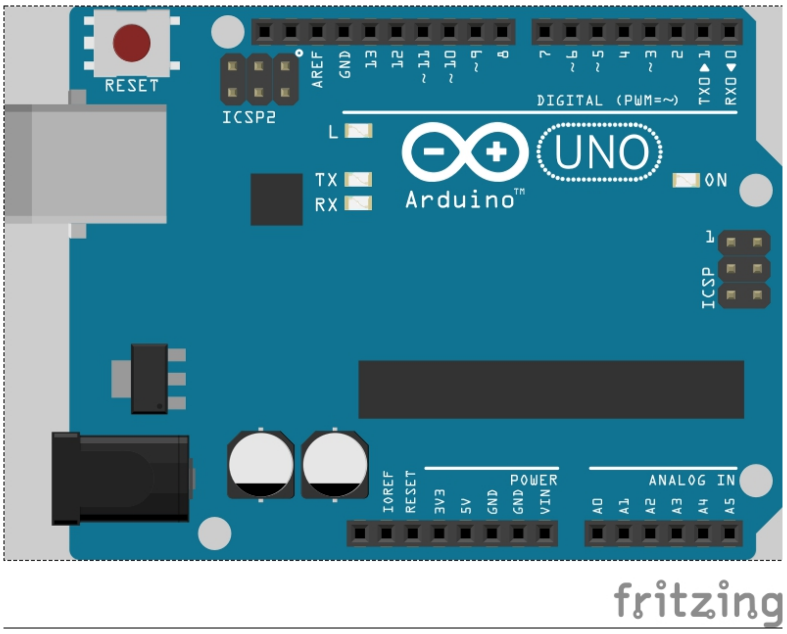

Figure 1: An Arduino Uno

During program development, the Arduino is usually powered via the USB connector. USB connection is also used to upload programs and for the serial connection between the Arduino and the host computer. On the bottom of the left side there is a power connector for an input voltage between 7 and 12V used when there is no host computer connected.

The original Arduino Uno operates at 5 volts, a number of its clones operate at 3.3 volts. This is especially important for the I/O pins. A high output voltage of a microprocessor running at 3.3V is only 3.3V instead of 5V. Some microprocessors running at 3.3V can be permanently damaged by a 5V input signal. Hence, the voltages must always be carefully checked before an Arduino is connected to some peripheral.

Since the Arduino is an open hardware standard there are many clones and variants of the system available. Some of them are rather cheap, often less than 10 €. A number of these variants contain additional hardware, like Bluetooth or WiFi modules.

In our lab we use the Iteaduino BT, which is basically an Arduino Duemilanove extended with a HC-05 Bluetooth module. We have version 1.0 as well as version 1.1 boards.

Near the mini-USB port there is a switch to operate this microprocessor either at 3.3V or at 5V. In 5V mode it is compatible with most Arduino shields. During ordinary Arduino operation, there should be no jumpers at the pins labeled `B A` above the HC-05 module.

There are some other types of Arduino's in the lab, see our wiki for more information. Apart from the genuine Arduino UNO's the are cheap DCCduino's, see Figure 3. Be

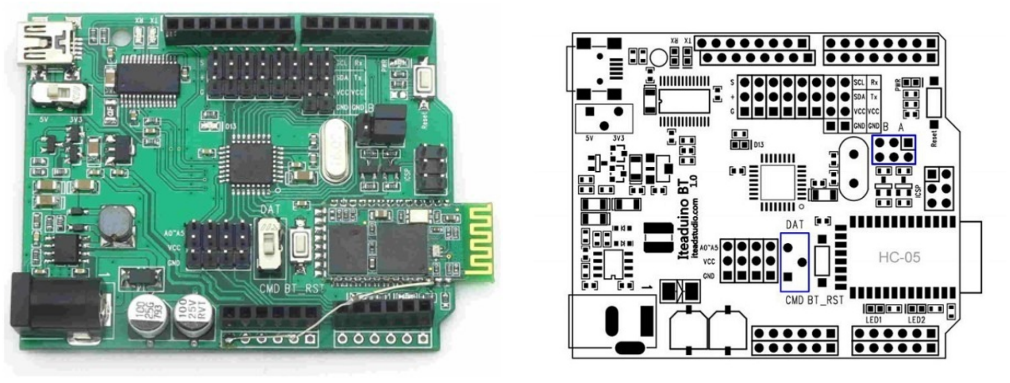

Figure 2: An Iteaduino BT

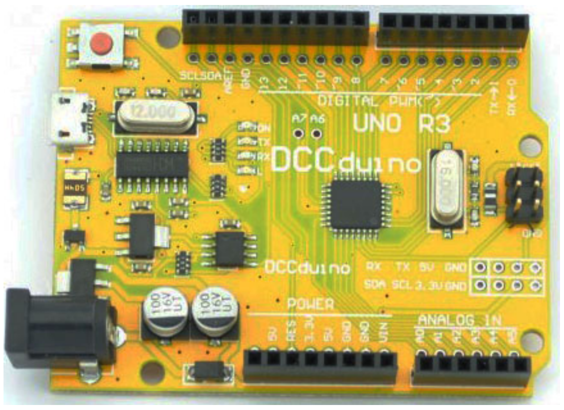

Figure 3: A DCCduino UNO R3

aware that you need a special driver library for the USB CH340 chipset on your computer. This can be downloaded from `http://www.wch.cn/download/CH341SER_ZIP.html`, or see `http://sparks.gogo.co.nz/ch340.html`. For a Mac the driver can be found at `http://www.wch.cn/download/CH341SER_MAC_ZIP.html`. On Linux the driver is usually available. If necessary, it can be downloaded from `http://sparks.gogo.co.nz/assets/_site_/downloads/CH340_LINUX.zip`.

1.2 Arduino Shields

A large part of the success of the Arduino is due to the availability of shields. These are printed circuit boards with connectors that are plugged into the headers of the Arduino board. Usually the shields have identical headers as the Arduino boards such that shields can be stacked. Shields provide additional hardware. Using shields the Arduino hardware can be expanded without soldering iron and even without breadboard or additional wires.

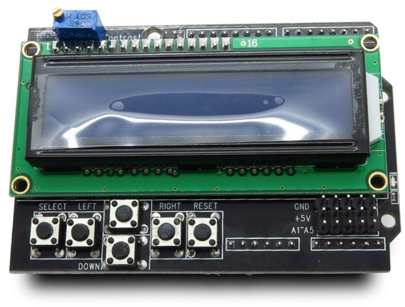

Figure 4: An Itead Studio LCD1602 key shield with two rows of 16 characters.

In this exercise we will use a shield with a LCD shield, five buttons for user defined input, and pins to access the analog I/O pins of the Arduino. All buttons are connected with a set of resistors to pin A0. By reading the input voltage at pin A0 we can figure out if a button is pressed and which one is pressed. The signals at pin D8, D9, D4, D5, D6, and D7 control what is displayed on the LCD.

Shields come typically with a library that contains a class to control this shield in the software.

1.3 Arduino Software

There is a free open source Arduino IDE that can be downloaded for Windows, Mac OS x, and Linux from `https://www.arduino.cc/en/Main/Software`. It can be used with any Arduino board as well as several other microprocessor systems. For a successful application, it is essential to choose the right kind of board in the tools menu and the proper USB-port connecting the board and the host computer. See `https://www.arduino.cc/en/Guide/HomePage` for detailed instructions and troubleshooting.

The Arduino does not run an operating system and has a very limited runtime system supporting your program. This has as advantage that your program is in full control over the system. Your program can directly control the I/O pins and other parts of the microcontroller. The disadvantage is that there is limited runtime support. Your program determines completely what is executed in each clock cycle of the Arduino.

The Arduino is programmed in a dialect of C++. Each Arduino program should provide at least the functions `void setup ()` and `void loop ()`. The `setup` is executed when you power up to the board, after uploading a new program to the board, and after pressing the `reset` button. Next, the `loop` function is executed over and over again as long as the microprocessor is running your program. When there is nothing to be done during `setup` or in the `loop` the corresponding function has an empty body.

1.3.1 Hello World

The "hello world" example for the Arduino blinks the onboard LED connected to digital pin 13. The pins of the Arduino can be used as input and output. It is required to set the pin in the right input–output mode before using it in that mode. The `setup` function sets pin D13, connected to the onboard LED, in output-mode. In the `loop` function we make this output high and low. After each write operation we wait 1000 milliseconds with the `delay` function.

```
void setup() {
  pinMode(13, OUTPUT);                  // initialize pin 13 as output
}

void loop() {
  digitalWrite(13, HIGH);               // switch LED on (HIGH voltage)
  delay(1000);                          // wait 1000 ms, i.e. one second
  digitalWrite(13, LOW);                // switch LED off (LOW voltage)
  delay(1000);                          // wait a second
}
```

1.3.2 Hello World without `delay`

Since there is no operating system, there are no other threads or programs on the Arduino. This implies that the `delay`(1000) call blocks program execution for one second. For more advanced programs it is better to prevent blocking by long delays. By introducing some state variables and looking repeatedly at the clock with `millis()`, returning the milliseconds since startup, the use of `delay` can easily be circumvented.

```
#define DELAY 1000
boolean ledOn = false;                  // status of LED
long lastTime = 0;                      // last status switch

void setup() {
  pinMode(LED_BUILTIN, OUTPUT);
}

void loop() {
  if (millis() - DELAY > lastTime) {    // time to change?
    ledOn = not ledOn;
    digitalWrite(LED_BUILTIN, ledOn );
    lastTime += DELAY;
  }
}
```

This way of programming makes it much easier to execute multiple tasks on the Arduino each at its own speed.

1.4 Serial Communication

The Arduino contains a serial port that can easily be accessed from the Arduino IDE as the outside world of the microprocessor. There is a class `Serial` controlling this serial connection on the Arduino side. The communication uses the USB-cable to power the board. Via the same connection a new program can be uploaded to the Arduino.

```
void setup() {
  Serial.begin(9600);              // open the serial port at 9600 bps
  Serial.println("Hello World!"); // print the message with a newline
}

void loop() {
}
```

See `https://www.arduino.cc/en/Reference/Serial` for a description of this class.

Most Arduino variants contain two LED indicating data traffic over the serial connection. The LED labeled `TX` blinks when the Arduino is **T**ransmitting data, and the `RX` blinks when the board **R**eceives data.

Since there are very limited ways to see what an Arduino board executing, blinking the LED on D13 and sending trace information to the serial port are heavily used to debug Arduino programs.

The next example shows how we can read input from the serial port and use this to control the LED on pin 13.

```
#define LED_PIN 13

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

void loop() {
  if (Serial.available() > 0) {
   char c = (char)Serial.read();
   switch (c) {
    case '1':
    case 'i':
    case 'I':
      digitalWrite(LED_PIN, HIGH);
      Serial.println("In");
      break;
    case '0':
    case 'o':
    case 'O':
      digitalWrite(LED_PIN, LOW);
      Serial.println("Out");
      break;
    }
    while (Serial.available() > 0) {
     Serial.read();
    }
    delay (250);
  }
}
```

Note that all methods of `Serial` are static. There is no `Serial` object constructed, all methods used belong to the class instead of to an object. This is possible in this situation since we assume that there is only one serial port.

1.5 The `LiquidCrystal` class

One of the most use shields contains a LCD of two lines of 16 characters each. The corresponding library is included by `#include <LiquidCrystal.h>`. A LCD object is made by the constructor `LiquidCrystal lcd(8,9,4,5,6,7)`, see `https://www.arduino.cc/en/Reference/LiquidCrystal` for a complete description. The arguments in this constructor indicate the pins used by this Arduino shield. Various brands of shields use different pins, and even a different number of pins. Hence, we should indicate the pins used in the constructor although it is fixed for a given type of shield. Since there can be several LCD screens connected to a single Arduino, we need to create a real `LiquidCrystal` object here (in contrast with the `Serial` class above).

A simple program that shows the message `Hello World!` on the display and scrolls it back and forth is:

```
#include <LiquidCrystal.h>                        // add LCD library
#define STEPS 10
int pos = 0;                                      // scroll steps done
int inc = 1;                                      // step inc
LiquidCrystal lcd = LiquidCrystal(8,9,4,5,6,7);   // define lcd object

void setup() {
  lcd.begin(16, 2);                               // set LCD size
  lcd.print("Hello world!");                      // display message
}

void loop() {
  if (inc > 0)                                    // scroll right?
    lcd.scrollDisplayRight();                     // scroll one step right
  else                                            // scroll left
    lcd.scrollDisplayLeft();                      // scroll one step left
  step();                                         // count step
  delay(400);
}
```

```
void step() {
  pos += inc;                                    // increment steps
  if (pos < -STEPS || pos > STEPS)               // outside bounds?
    inc = -inc;                                  // turn around
}
```

Part of the message is scrolled outside the display, but remembered by the `LiquidCrystal` object `lcd`. Note that this program uses the variables `pos` and `inc` to store the state between subsequent calls of `loop()`. This is a very common pattern in Arduino programs. This example also uses a simple user define function.

Instead of the given definition of the `LiquidCrystal` object, you will often encounter the statement `LiquidCrystal lcd(8,9,4,5,6,7);` as C++ shorthand for the construction of the object `lcd`. A Java programmer will miss the keyword **new** in this object definition. C++ has a keyword **new**, but that creates a *dynamic* object. Such a dynamic object will consume memory until it is explicitly deleted in the program. We hardly ever use these kinds of objects in an Arduino due to the limited heap space. Note that objects are usually constructed outside the `loop` function. Especially creating dynamic objects inside the `loop` functions is a potential source of memory problems. Moreover, an object definition with a **new** yields a pointer to an object instead of the object itself. For these reasons the keyword **new** is not encountered often in Arduino programs.

1.5.1 Buttons on the LCD-shield

Many LCD-shields are equipped with buttons for user input. Typically there are 6 buttons. The rightmost button is the reset button of the Arduino board. It is convenient to duplicate this button to the shield since the original reset button is unreachable through the placement of the shield.

To safe I/O pins the other five buttons are connected to a single analogue input pins: A0. Via a network of resistors, called a voltage ladder, we can determine the button pressed.

In a perfect world the Analog-Digital-Convertor of pin A0 will produce the listed numbers when a button is pressed. In the real world there are tolerances for the resistors and inaccuracies in the AD-converter, this result in slightly different readings from input A0 when a button is pressed. A typical program tests a value somewhere between the given numbers to determine the button pressed.

```
#include <LiquidCrystal.h>
#define KEY_COUNT 5

int keyLimits [KEY_COUNT+1]      = {50, 190, 380, 555, 790, 1024};
char keyNames [KEY_COUNT+1] [10]
  = {"Right ", "Up    ", "Down  " , "Left  " , "Select" , "No key"};
LiquidCrystal lcd = LiquidCrystal(8, 9, 4, 5, 6, 7);

void setup() {
  lcd.begin(16, 2);
}
```

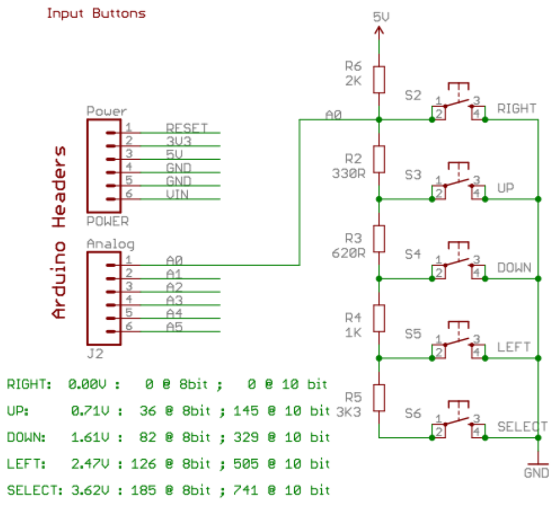
Input Buttons
5V
R6 2K
S2
RIGHT
A0
Power
RESET
3V3
5V
GND
GND
VIN
POWER
Arduino Headers
Analog
A0
A1
A2
A3
A4
A5
J2
R2 330R
S3
UP
R3 620R
S4
DOWN
R4 1K
S5
LEFT
R5 3K3
S6
SELECT
GND
RIGHT: 0.00V : 0 @ 8bit ; 0 @ 10 bit
UP: 0.71V : 36 @ 8bit ; 145 @ 10 bit
DOWN: 1.61V : 82 @ 8bit ; 329 @ 10 bit
LEFT: 2.47V : 126 @ 8bit ; 505 @ 10 bit
SELECT: 3.62V : 185 @ 8bit ; 741 @ 10 bit

Figure 5: The voltage ladder for the buttons of the LCD-shield.

```
void loop() {
  int val = analogRead(A0);
  lcd.setCursor(0, 0);
  lcd.print(val);
  lcd.print(" on A0    ");
  for (int i = 0; i <= KEY_COUNT; i += 1) {
    if (val < keyLimits[i]) {
      lcd.setCursor(0, 1);
      lcd.print(keyNames[i]);
      break;
    }
  }
  delay(500);
}
```

1.6 The `Servo` class

In the same style there is a `Servo` library to control servos. A servomotor is a rotary actuator that allows for precise control of angular position by pulse-width modulation, PWM. A servo consists of a suitable motor coupled to a sensor for position feedback. Pulse-width modulation is a modulation technique used to encode a message into a pulsing signal. Here the duration of the pulse on the input wire of the servo controls its position.

Fortunately the Arduino offers ample support for servo control. The digital pins 3, 5, 6, 9, 10, and 11 provide 8-bit PWM output with the `analogWrite` function. Even better, the `Servo` library supports up to 12 motors on most Arduino boards. On boards other

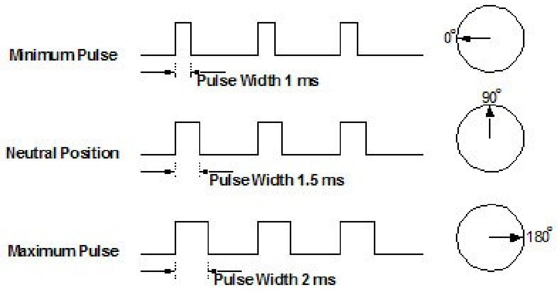

Figure 6: The relation between PWM signals and servo positions.

than the Mega, use of the library disables analogWrite() (PWM) functionality on pins 9 and 10, whether or not there is a servo on those pins.

Servomotors have three wires: power, ground, and signal. The power wire is typically red, and should be connected to the 5V pin on the Arduino board. The ground wire is typically black or brown and should be connected to a ground pin on the Arduino board. The signal pin is typically yellow, orange or white and should be connected to a digital pin on the Arduino board. Note that servos draw considerable power, so if you need to drive more than one or two, you'll probably need to power them from a separate supply (i.e. not the +5V pin on your Arduino). Be sure to connect the grounds of the Arduino and external power supply together. The Arduino can power the small servos in the lab, no additional power supply is needed there.

The next program sweeps the servo between a minimum and a maximum number of degrees. In theory minimum and maximum should be 0 and 180 degrees respectively. In practice the range is often a bit smaller do to physical inaccuracies.

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

#define MAX_ANGLE 170
#define MIN_ANGLE 10
#define TICK 25

Servo servo = Servo();                  // create servo object
int pos = (MAX_ANGLE + MIN_ANGLE) / 2;  // servo position
int step = 1;                           // the change in servo angle
long time = 0;                          // last state change

void setup() {
  servo.attach(A5);                     // indicate control wire to servo object
}

void loop() {
  if (millis() - TICK > time) {         // time to change?
    time += TICK;
    if (pos > MAX_ANGLE || pos < MIN_ANGLE) {
      step = -step;                    // change direction
    }
    pos += step;
    servo.write(pos);                  // set servo position in degrees
  }
}
```

Since there can be many servos in a single program we need to construct a servo objects instead of using static methods.

1.7 Ultrasonic Distance Sensor

Not all peripherals for the Arduino are mounted on shields. When there are only a few wires needed, or the peripheral must be positioned away from the Arduino, one often uses either a breadboard, or just a few jumper wires.

A typical example is an ultrasonic distance sensor. Placement of the sensor is essential to measure a distance in the right direction. Applications like autonomous cars and robots have often several of these ultrasonic sensors to form an image of the surroundings.

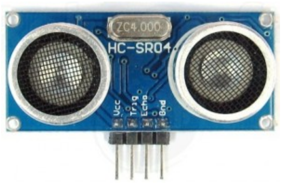

Figure 7: A HC-SR04 ultrasonic sensor.

The sensor can send a small burst of ultrasonic sound. Most solid objects reflect this sound. The sensor can detect the reflected sound. This is depicted in Figure 8. The distance between sensor and object can be computed from the reflection time. Traveling to the object and back gives a factor of 2. From physics it is known that sound takes 29.1 ms to travel one centimeter.

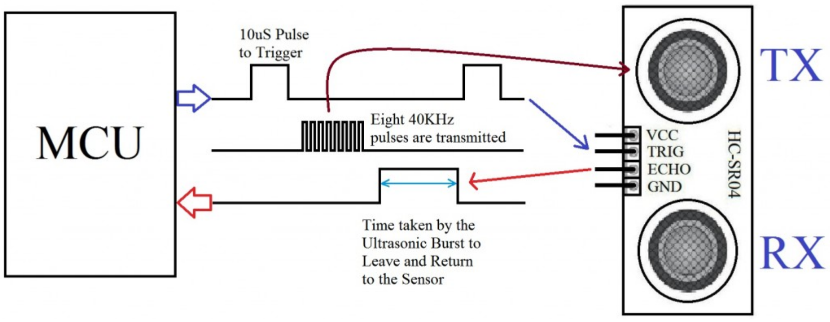
MCU
10uS Pulse to Trigger
Eight 40KHz pulses are transmitted
Time taken by the Ultrasonic Burst to Leave and Return to the Sensor
VCC
TRIG
ECHO
GND
HC-SR04
TX
RX

Figure 8: Working of the ultrasonic sensor.

The electronic connections between the ultrasonic sensor and the Arduino board are depicted in Figure 9. The colors of the jumper wires are of course arbitrary. We try to stick to the convention that Vcc, the +5 volt line, is red and the ground in black in the drawings.

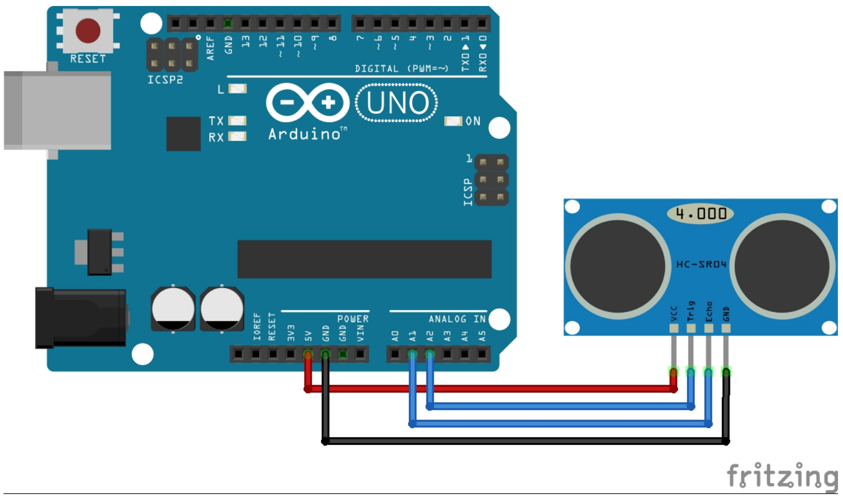

Figure 9: Connecting an ultrasonic sensor to an Arduino board.

In the corresponding code we define names for the used pins in the code such that it is easy to switch pins. During setup we set the baud rate of the serial connection and the I/O mode of the pins used. In the `loop` we generate an ultrasonic burst of 10 microseconds. Next we measure the duration of the echo time with the library function `pulseIn` and write the results over the serial connection.

```
#define trigPin A2
#define echoPin A1

void setup() {
  Serial.begin (9600);
  pinMode(trigPin, OUTPUT);
  pinMode(echoPin, INPUT);
}

void loop() {
    digitalWrite(trigPin, HIGH);
    delayMicroseconds(10);
    digitalWrite(trigPin, LOW);
    int duration = pulseIn(echoPin, HIGH);
    Serial.print(duration);
    Serial.println(" ms");
    Serial.print((duration / 2) / 29);
    Serial.println(" cm");
    delay(1000);
}
```

There is a library called `NewPing.h` providing convenient abstractions for ultrasonic sensors, see `https://bitbucket.org/teckel12/arduino-new-ping/wiki/Home`.

1.8 433 MHz Radio Communication

Many new devices have wireless remote control. A large part of these controls use radio communication instead of infrared light communication to avoid the necessity of visual contact between sender and receiver. Typical applications are remote access for cars, remote controlled sockets, remote temperature sensors, wireless doorbells etc.

A large fraction of these radio based remote controls use am amplitude-modulated signal at 433Mhz. This frequency band is globally available and license-free for low power transmitters. Radio signals at this frequency is ideal for wireless sensor networking applications since it penetrates concrete and water, but also has the ability to transmit/receive over long ranges without requiring a large power draw on a battery. The low input current of typical tag configurations allows for battery powering on coin cell or thin film batteries.

There are senders and receivers available for the Arduino to build your own radio-controlled devices. These senders are cheap small and can operate with the power supplied by the Arduino. To increase the range of the senders it is possible to operate them on a higher voltage.

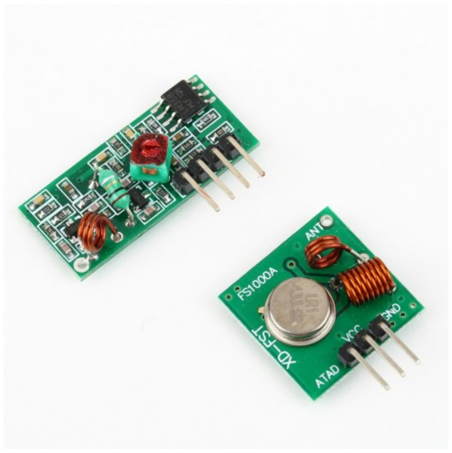

Figure 10: A 433 MHz receiver on the left top and a corresponding sender.

To operate a 433 MHz radio receiver we just need to connect the power, the ground and one of the data connections.

Using an analog pin of the Arduino it is possible to monitor the amplitude of the signal. A more sophisticated operation mode uses a digital pin and an interrupt handler that only is invoked when the receiver encounters a signal of sufficient strength. The `RCSwitch`

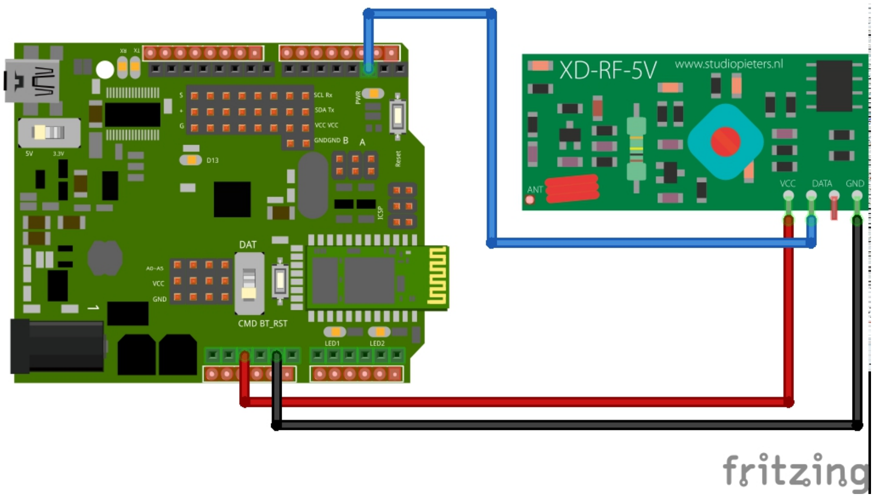

Figure 11: The 433 MHz receiver connections.

library from `https://github.com/sui77/rc-switch` implements just this. With the `enableReceive` method the Arduino will listen to incoming signals on the external interrupt pin (first parameter). Most Arduino boards have only two external interrupts: number 0 (on digital pin 2) and 1 (on digital pin 3).

```
#include <RCSwitch.h>

RCSwitch mySwitch = RCSwitch();

void setup() {
  Serial.begin(9600);
  Serial.println("RC receiver");
  mySwitch.enableReceive(0);  // Receiver on interrupt 0 => that is pin #2
}

void loop() {
  if (mySwitch.available()) {
    long value = mySwitch.getReceivedValue();
    if (value == 0) {
      Serial.print("Unknown encoding");
    } else {
      Serial.print("Received ");
      Serial.print( value );
      Serial.print(" / ");
      Serial.print( mySwitch.getReceivedBitlength() );
      Serial.print(" bit, protocol: ");
      Serial.println( mySwitch.getReceivedProtocol() );
    }
    mySwitch.resetAvailable();
  }
}
```

This program can be used to capture commands from a 433 MHz remote control. You will notice that receiving commands is not perfect. It seems that the remote control sends every now and then different commands when you press the same button. The signal detect in the majority of the cases is probably the correct signal. The newer "Klik Aan Klik Uit" systems use a different encoding, the sockets still understand the traditional signals from other remote controls.

The 433 MHz sender has three connections; the positive power supply, Vcc, the ground, GND, and a data port. This data pin can be connected to any Arduino output, we use pin A5 in the example below.

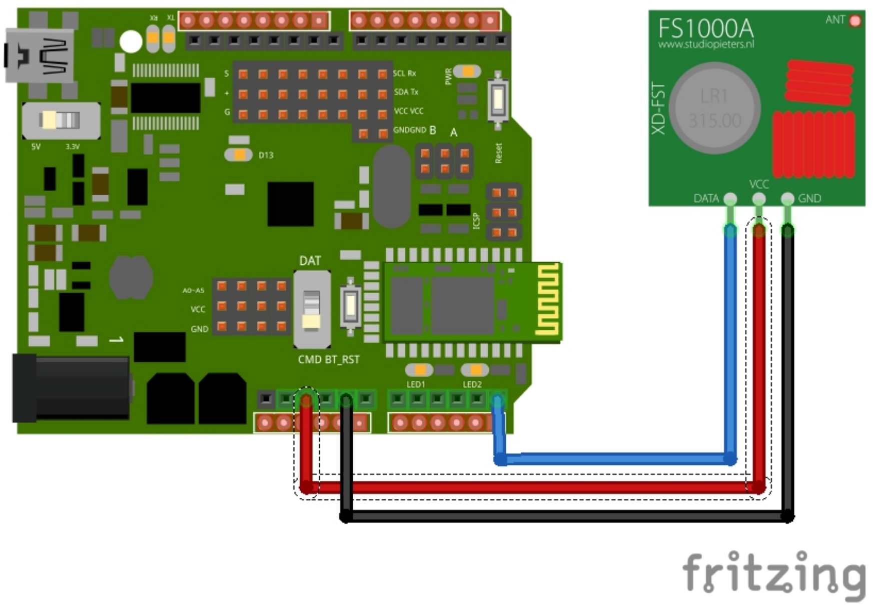

Figure 12: The 433 MHz sender connections.

In our program we use the codes found by the RC-receiver above. The program is very similar to blink. Instead of switching the LED on and off, this program switches the 433 MHz controlled socket on and off.

```
#include <RCSwitch.h>

#define OnCode 5587989
#define OffCode 5587988
#define CodeLength 24
#define SendPin A5

RCSwitch mySwitch = RCSwitch();
```

```
void setup() {
  pinMode(SendPin, OUTPUT);
  mySwitch.enableTransmit(SendPin);
  mySwitch.setProtocol(1);
}

void loop() {
  mySwitch.send(OnCode, CodeLength);
  delay(1000);
  mySwitch.send(OffCode, CodeLength);
  delay(1000);
}
```

More complex programs can be made based on these examples.

1.9 Bluetooth Communication

Bluetooth is a wireless technology standard for exchanging data over short distances. It uses short-wavelength UHF radio waves between 2.4 and 2.485 GHz. The Bluetooth system allows mobile phones to communicate with computers. There are various variants of this standard that covers the electrical signals as well as higher levels of a protocol, known as the Bluetooth protocol stack. Today the most used variant is Bluetooth 4, also known as *Bluetooth low energy*. The good news is that a Bluetooth connection can be used very similar to a serial port, we can just send a message to a Bluetooth connection and receive it on the other side.

There are Bluetooth shields for the Arduino. The Iteaduino BT used in the lab has on onboard Bluetooth communication unit. We can use this for wireless communication with mobile phones and computers. The HC05 module implements the Bluetooth protocol stack.

There are two modes of communication with the HC05 module; one mode to configure the module as Bluetooth device, and the mode uses the module to send and receive messages. The jumpers on the six pins above determine the mode the HC05 module together with the switch located just left of it.

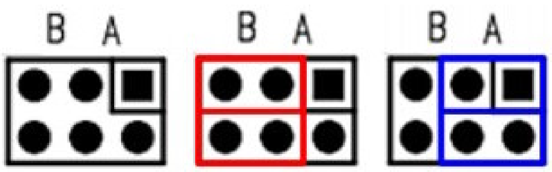

Figure 13: Bluetooth control jumpers on the Iteaduino BT.

There are three modes of operation relevant for us:

1. without any jumpers, or jumpers not connecting any pins, the HC05 module is not connected. It is safe to put the switch in `DAT` mode. This is the leftmost

configuration in Figure 13. Use this setting for ordinary Arduino operation of the Iteaduino BT.

2. With two jumpers connecting the central pins to their left neighbors, labeled B, the serial communication of the board is via the Bluetooth module. This configuration is in the middle of Figure 13. The switch must be in the `DAT` position, for `DATA`. Any `Serial.print` command executed will send its data via the Bluetooth connection. Serial read commands will also work on the Bluetooth connection. Before we can actually send and receive data the module must be paired with the other Bluetooth module in the communication, either a mobile phone or a computer.

3. When the jumpers are in position A, the rightmost configuration in Figure 13, and the switch is in position `CMD`, for `COMMAND`, the Bluetooth module itself can be configured via the serial port of the board. This works only when the program on the board does not use the serial port, and the serial port is set to 38400 baud and sends newline and carriage return, `BOTH NL` & `CR`. The easiest way to avoid that the program on the Arduino does not interference with this use of the serial monitor is to upload the `blink` program, do not forget to remove the jumpers before uploading this program.

   In this mode the Bluetooth mode can be configured by `AT` commands entered in the serial monitor of the Arduino IDE. The most important command is `AT+ORGL` to restore the default settings. The answer of the module in the serial monitor should be `OK`.

   See `http://wiki.iteadstudio.com/Serial_Port_Bluetooth_Module_(Master/Slave)_:_HC-05` for an overview of `AT` commands.

There is some confusion on the internet over the position of the jumpers. Apparently, it has changed over various versions.

In Bluetooth communication there is of course a second devices needed. There are many options here ranging from simple to pretty advanced. The App `Bluetooth Terminal` for Android is able to send and receive string via the Bluetooth connection, see `https://play.google.com/store/apps/details?id=Qwerty.BluetoothTerminal`. Similar programs are available for other operating systems. Our simple program reading the serial port from Section 1.4 should allow us to control the LED via Bluetooth.

1.10 WiFi Communication

The Arduino board is not powerful enough to implement the WiFi protocol. Hence, any WiFi-shield for the Arduino contains its own microprocessor. This works very similar to Bluetooth modules. A recent module is called ESP8266.

In fact this microprocessor is far more powerful than the ATmega328 microprocessor of the Arduino board. Moreover, the ESP8266 has some spare pins that can be used very similar to Arduino I/O pins. For this reason people start using it as a replacement for the Arduino instead of only as the WiFi module of an Arduino board. However, it is important to notice that the ESP8266 uses 3.3 volt instead of the 5 volt of the Arduino. In fact the ESP8266 is usually damaged when we apply 5 volt to one of its I/O pins. The ESP8266 can be programmed from the Arduino IDE, see `https://github.com/esp8266/Arduino` for details. For the time being we have to use version 1.6.5 of the Arduino IDE instead of the last version. With the last version thing would most likely not work.

The `Nodemcu` is a ESP8266 microprocessor placed on an Arduino like board, see `http://nodemcu.com`. The board is smaller and is not compatible with Arduino shields in shape and voltage. Nevertheless, we can use this quite convenient to transmit 433MHz signals. Figure 14 depicts the necessary connections. Note that we use the 5 volt input as Vcc for the sender instead of the onboard 3.3 volt to increase the range of the sender. This board contains jet another USB interface chip which needs its own software driver. This can be downloaded from `https://www.silabs.com/products/mcu/Pages/USBtoUARTBridgeVCPDrivers.aspx`.

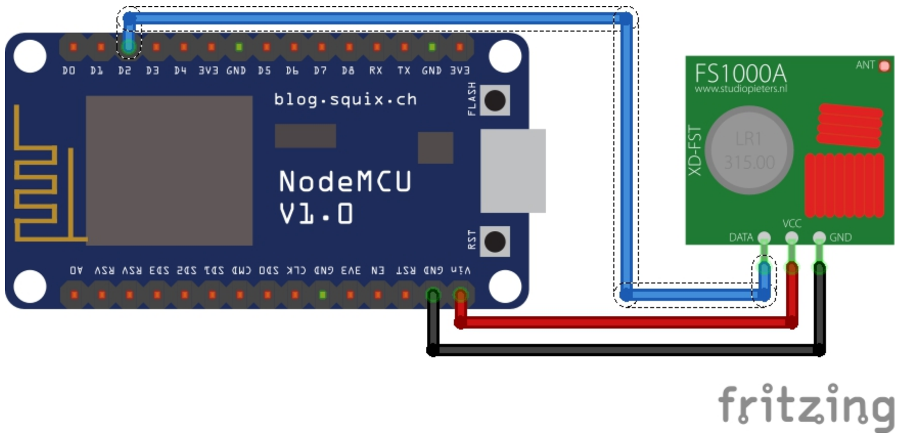
D0 D1 D2 D3 D4 3V3 GND D5 D6 D7 D8 RX TX GND 3V3
blog.squix.ch
FLASH
NodeMCU V1.0
RST
A0 RSV RSV SD3 SD2 SD1 CMD SDO CLK GND 3V3 EN RST GND Vin
FS1000A
www.studiopieters.nl
ANT
XD-FST
LR1 315.00
DATA
VCC
GND
fritzing

Figure 14: Nodemcu 433 MHz sender connections.

Using this board it is rather simple to make you own WiFi controlled device. Our example generates a webpage with a single button. Pressing the button switches the state of the onboard LED.

```
#include <ESP8266WiFi.h>
#include <ESP8266WebServer.h>
#include <ESP8266mDNS.h>

MDNSResponder mdns;
const char* ssid = "....";        // insert name of local WiFi network
const char* password = "....";    // insert the password of the WiFi network
#define LISTEN_PORT 80
ESP8266WebServer server = ESP8266WebServer(LISTEN_PORT);
String webPage = "";
```

```
bool ledOn = false;

void setup() {
  Serial.begin(115200);         // the serial speed of the Nodemcu
  pinMode(D0, OUTPUT);          // the LED of the Nodemcu
  digitalWrite(D0, ledOn);      // the actual status is inverted

  webPage += "<h1>WiFi LED control</h1>";
  webPage += "<p><b>Press me <a href=\"button\">";
  webPage += "<button style=\"background-color:blue;color:white;\">";
  webPage += "LED</button></a></b></p>";

  // make the WiFi connection
  WiFi.begin(ssid, password);
  Serial.println("Start connecting.");
  while (WiFi.status() != WL_CONNECTED) {
    delay(500);
    Serial.print(".");
  }
  Serial.println("");
  Serial.print("Connected to ");
  Serial.print(ssid);
  Serial.print(". IP address: ");
  Serial.println(WiFi.localIP());

  if (mdns.begin("esp8266", WiFi.localIP())) {
    Serial.println("MDNS responder started");
  }

  // make handlers for input from WiFi connection
  server.on("/", [](){            // refresh
    server.send(200, "text/html", webPage);
  });

  server.on("/button", [](){      // button pressed
    server.send(200, "text/html", webPage);
    ledOn = !ledOn;
    Serial.print("led ");
    Serial.println(ledOn);
    digitalWrite(D0, ledOn);
    delay(1000);
  });

  // start the server for WiFi input
  server.begin();
  Serial.println("HTTP server started");
}

void loop() {
  server.handleClient();
}
```

When we open a browser at the IP address listed at the serial monitor we should get a very simple webpage with a single button as specified in the program. The status of the onboard LED, here connected to pin D0, should switch each time the button on the webpage is pressed.

There are umpteen other libraries that can be used to control the esp8266 and associated webpages. Google will give you many hits. Make sure to use only Arduino controlled versions in this exercise.

2 Assignment

In this exercise you have to make a controller for two radio-controlled sockets. Both sockets should go on when you press the `UP` button on a LCD shield. The `DOWN` button should switch the sockets off. The `LEFT` button switches one socket on and the other off. On pressing the `RIGHT` button the other socket goes on and the first one off. The LCD displays the status of the sockets.

It will be necessary to record the commands accepted by the sockets with the RF-receiver discusses above. When the LCD shield is placed on the Arduino it is next to impossible to connect the 433 MHz receiver properly. Do not try this.

In addition one of the sockets switches on when the ultrasonic sensor detects on object within 100 cm. When no object is detected for more than one (or any positive number you like) minute the sockets go back to the status given by the keys.

Add a servo in such a way that the ultrasonic sensor scans its surroundings.

Extend your controller with a Bluetooth or a WiFi interface. Since the LCD shied is not compatible with the Nodemcu, it is not possible to include this in one program.

2.1 Deliverables

Submit the code of your solution to blackboard.

- Deadline 5 February 23:59.
- Include your name and the name of your partner and the student numbers.
- Include the type of sockets and the associated codes used.
- Include in comments sufficient information about shields and connections made to execute your code.