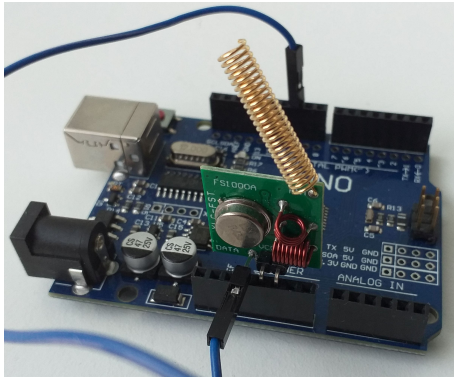
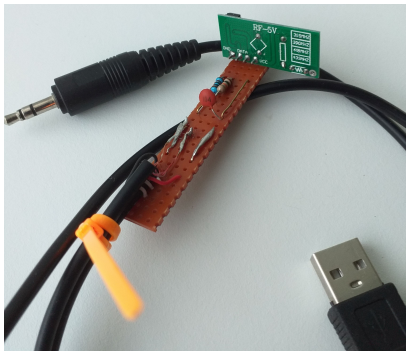
Programming remote controllable (RC) sockets.

For our project, to switch a boiler socket “on” when the solar panels are delivering enough power, I wanted to add RC sockets to our system. This would make running cables though the house unnecessary: Just put a RC socket between a wall socket and the user.
Be sure to **check their power rating** beforehand. Some RC sockets can only handle 1000 Watts.

There are some other webpages describing a way to program the processor and there exists a “RCSwitch library” for the Arduino microprocessor. As I couldn't get this to work for our sockets I included this more basic way which doesn't use a specific library.

To operate RC sockets without the remote you need a small 433 MHz transmitter-shield to send the signal. You also have to figure out the signal for each individual command. For this a 433MHz receiver on your computer's sound card, together with the program Audacity can be used:



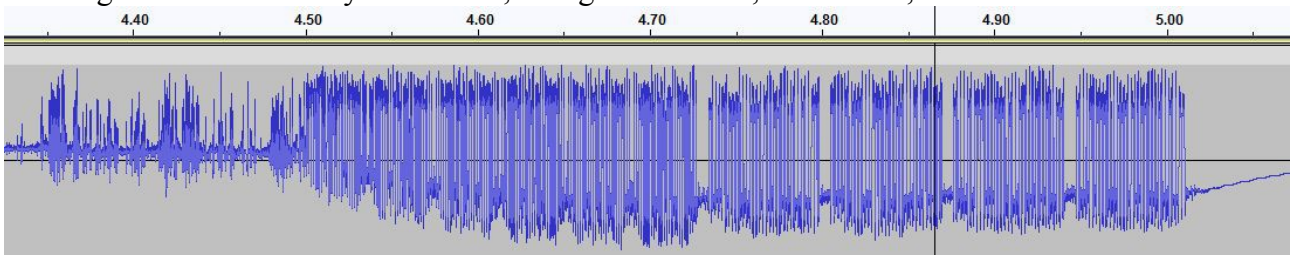


Transmitter (left) connected to the 5V and GND of the |Arduino, the angled data header pin is straightened to evade the female header of the Arduino and is lead to digital output 10 at the other side.
Receiver (right) The 5V and GND are supplied via an USB connection, the data and GND connect to an audio plug to connect to your computer's sound-card. The data signal from the receiver is made weaker with a 1k-10k resistor bridge.
Transmitter and receiver are mostly sold together for a few Euro's.

Finding the commands

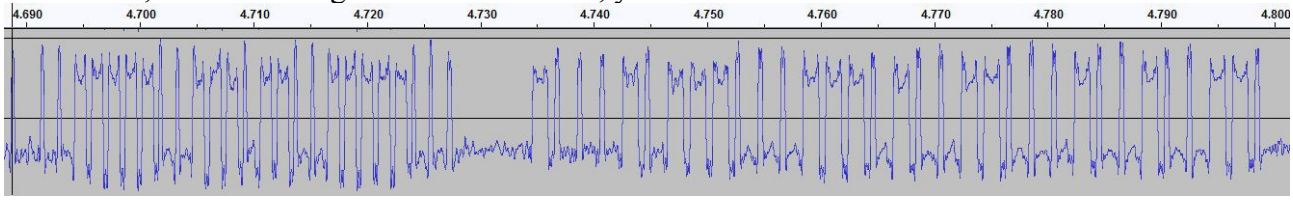
If you run Audicity and push the remote of the RC Sockets, you will receive, apart from a lot of disturbance, the given remote commands. They will be audible and visible on the recording.

Looking at the data send by the remote, a single command, in our case, looks like this:

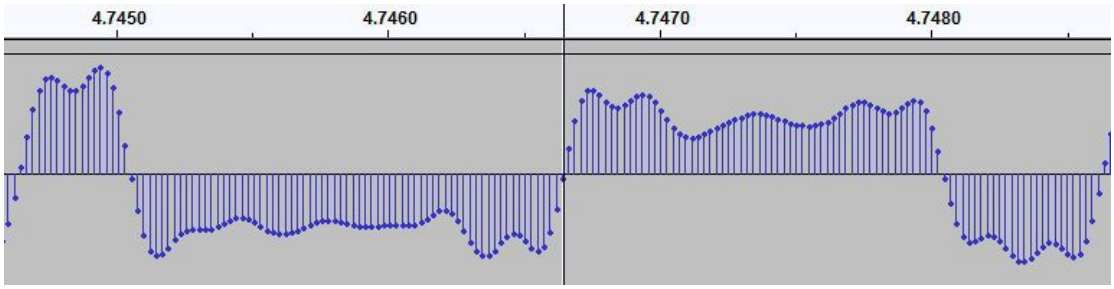


In this case it consists of a pattern of 24 bits, which is send 6 times followed by a pattern of 32 bits, which is send 4 times.

Zoomed in, near the change from 24 to 32 bits, you can also notice the time difference:



Zoomed in to a level that the sample points are visible, a “LOW” and “HIGH” bit look like this:



The standard sample rate of Audacity is 44100 Hz. So, for example, 44 sample points take 44 / 44100 = 998 µ seconds;
To simulate the signal later in the program I define the times of the LOW and HIGH for each bit type, and times of the HIGH/LOW part at the start of the patterns. The code sequences are stored in two multi arrays, one for the 24 bits pattern and the other for the 32 bits pattern.

In the program it looks like this:
// delay times (HIGH or LOW situations) in µ seconds:

```
#define Bit24LowUp 295          //13 samplepoints
#define Bit24LowDown 1179       //52 samplepoints
#define Bit24HighUp 998         //44 samplepoints
#define Bit24HighDown 476       //21 samplepoints
#define Bit32LowUp 408          //18 samplepoints
#define Bit32LowDown 1610       //71 samplepoints
#define Bit32HighUp 1429        //66 samplepoints
#define Bit32HighDown 590       //26 samplepoints
#define StartDelay24LowUp 400
#define StartDelay24LowDown 2250
#define StartDelay32LowUp 408
#define StartDelay32LowDown 7188
```

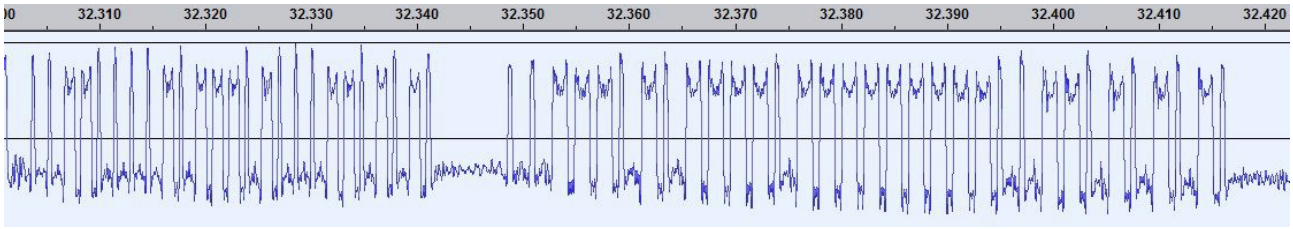
Having 3 sockets, each with a “ON” and “OFF”, the 6 commands and arrays look like this:

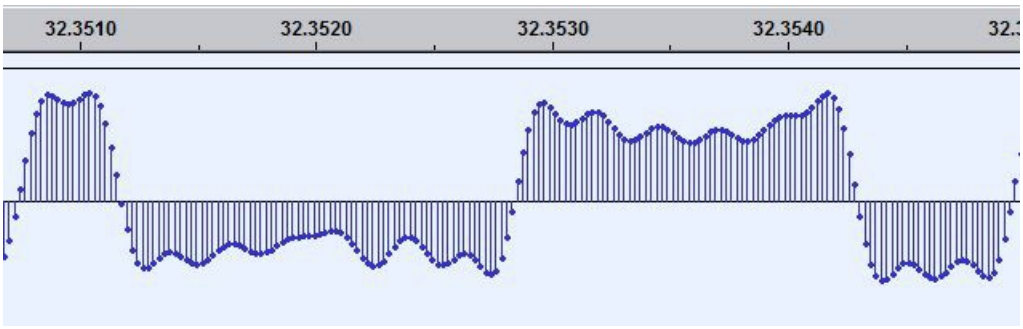
```
bool S24[6][24] =   {{0,0,1,1,1,0,1,1,1,1,0,0,0,0,1,1,0,1,1,0,1,1,0,0},   //  A_ON
                     {0,0,1,1,1,1,1,1,0,0,1,1,1,0,1,1,0,0,1,1,1,1,1,0,0},  //  A_OFF
                     {0,0,1,1,0,0,0,0,1,0,1,1,1,0,1,0,0,0,1,1,0,1,0,1},     // B_ON
                     {0,0,1,1,0,1,1,1,0,0,0,0,1,1,0,1,0,1,0,0,0,1,0,1},     // B_OFF
                     {0,0,1,1,0,1,1,0,1,1,1,0,0,0,0,1,1,0,1,1,1,1,1,0},     // C_ON
                     {0,0,1,1,0,1,0,1,0,1,1,0,1,0,0,0,0,0,0,1,1,1,1,0}};    // C_OFF

bool S32[6][32] =   {{1,1,1,1,0,0,0,1,0,1,0,1,1,1,1,1,1,1,1,1,1,0,1,0,1,1,0,1,0,0,1,1},    // A_ON
                     {1,0,0,0,1,0,1,1,1,0,0,0,1,1,1,0,1,0,0,1,1,0,0,0,1,0,0,1,0,0,1,1},   // A_OFF
                     {0,0,1,1,1,0,1,0,1,1,1,1,0,1,1,1,1,1,1,1,1,1,0,0,1,1,0,1,0,1,0,1},   // B_ON
                     {1,0,1,1,1,1,0,0,1,1,1,0,1,1,1,0,0,1,1,0,1,1,1,1,1,1,0,0,1,0,1,0,1},   // B_OFF
                     {1,1,1,1,1,1,1,1,1,1,0,1,1,0,0,1,1,0,1,0,1,0,1,0,0,1,1,1,1,0,1,1,0,1,1},   // C_ON
                     {1,1,0,1,0,0,1,1,0,0,1,0,0,0,0,1,1,0,0,0,0,0,0,0,1,1,0,0,1,1,0,1,1}};   // C_OFF
```

Apart from the bit patterns you should start each pattern with a long LOW part to make the patterns distinguishable. The timing of these LOW parts at the start is important and shouldn't deviate to much from the original.
During the running of the program, I call the same function to execute the operation of the sockets and only have to supply the required command number:

```
void switchRC(int commandNumber)
{
  int bitNumber;
  int bitPattern;
  for (bitPattern = 0; bitPattern < 6; bitPattern++)  // 24 bit part 6x
  {  // start with the extra long LOW part
    digitalWrite(pinRCdata, HIGH);
    delayMicroseconds(StartDelay24LowUp);
    digitalWrite (pinRCdata, LOW);
    delayMicroseconds(StartDelay24LowDown);
    for (bitNumber = 0; bitNumber < 24; bitNumber++)
    {
      if (S24[commandNumber][bitNumber] == HIGH)
      {
        digitalWrite (pinRCdata, HIGH);
        delayMicroseconds(Bit24HighUp);
        digitalWrite (pinRCdata, LOW);
        delayMicroseconds(Bit24HighDown);
      }
      else
      {
        digitalWrite (pinRCdata, HIGH);
        delayMicroseconds(Bit24LowUp);
        digitalWrite (pinRCdata, LOW);
        delayMicroseconds(Bit24LowDown);
      }
    }
  }
  for (bitPattern = 0; bitPattern < 4; bitPattern++)  // 32bits part 4x
  {  //start with  an extra long LOW:
    digitalWrite (pinRCdata, HIGH);
    delayMicroseconds(StartDelay32LowUp);
    digitalWrite (pinRCdata, LOW);
    delayMicroseconds(StartDelay32LowDown);
    for (bitNumber = 0; bitNumber < 32; bitNumber++)
    {
      if (S32[commandNumber][bitNumber] == HIGH)
      {
        digitalWrite (pinRCdata, HIGH);
        delayMicroseconds(Bit32HighUp);
        digitalWrite (pinRCdata, LOW);
        delayMicroseconds(Bit32HighDown);
      }
      else
      {
        digitalWrite (pinRCdata, HIGH);
        delayMicroseconds(Bit32LowUp);
        digitalWrite (pinRCdata, LOW);
        delayMicroseconds(Bit32LowDown);
      }
    }
  }
}
```

The computer simulated commands and bit patterns look like this:





They work fine for all the sockets.

program listings
https://www.bootprojecten.nl/solar-energy/programming-rc-sockets-433mhz

_240709_Solar_Switched_BoilerSocket.ino
Simple boiler socket switching

_240719_Solar_Switched_Boiler_Socket.ino
As above with 3-way switch “Always On”, ”Solar Regulated”, ”Always Off”

_240828_Solar_Switched_Sockets.ino
As above with RC Sockets switching along depending on settings

The programs are pretty much self explanatory and you can easily adjust the settings to your own situation and preferences.

An alternative to this system is to find RC sockets with a type of remote of which you can manipulate the buttons and use the transmitter of the remote. You will need a digital port for each command though and likely a lot of miniature mechanical fiddling. I tried earlier with ours but couldn't get it to work in a trustworthy way.

For feedback, questions or remarks you can mail in Dutch or English to

Jeroen Droogh bootprojecten@gmail.com

https://www.bootprojecten.nl/solar-energy/solar-power-regulated-socket

https://www.bootprojecten.nl/solar-energy/getting-data-from-a-solar-panel-inverter

https://www.bootprojecten.nl/solar-energy/solar-controlled-socket-shield-for-arduino