HILINE

Menu

Posted on 3/21/20

Android controlled lights

Project description

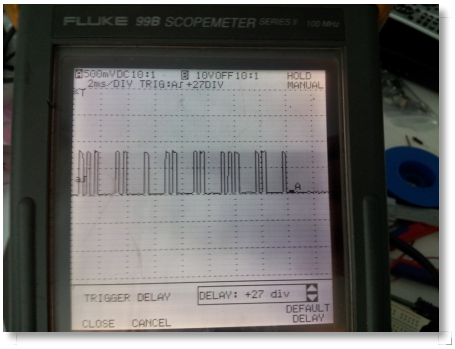
My goal is to control the KaKu switches with my phone through an Arduino or a Teensy. I currently have Remote Controlled power sockets from 'Klik-aan Klik-uit'. These can be switched on and off with the Remote Control. I thought it would be nice if could be done with the phone as well.

Klik-aan Klik-uit

KaKu stands for Klik-aan Klik-uit (click-on click-off). With this product you can switch the lights on and off with a remote contro It is a popular product in the Netherlands and available at Home depot stores. For my project I use the AYCT-102 remote control, a ACD-300 power socket and a AWMD-250 dimmer module. These use a new protocol compared with the older models. This new series can also send specific dim levels (only 16) to the sockets. The dimming module is built in behind a light switch. The down side of this dimmer module is when you turn on the light, it first goes to 100% and then to the dim level So if you set your light on 50% and you turn the light off and on again, it first goes to 100% and then to 50%. This is really stupid. According to the manual this is needed to support all types of lights. Therefore I don't recommend this module. There also seem to be modules who don't have this problem.

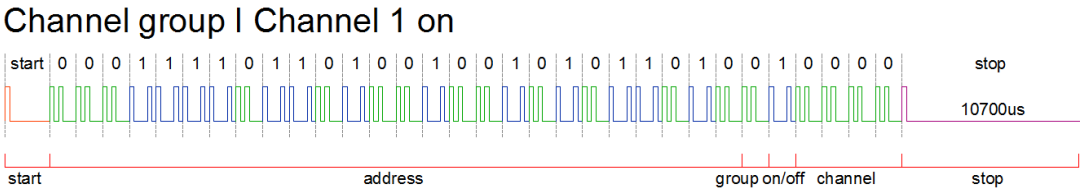
The downside of the product is, that it has no bi-directional communication. This means it only can send command from the remote control to the sockets and there is no return signal to check if confirm that the signal has arrived. Also the current state of a socket cannot be retrieved. If this is requested, you should use a product with a different protocol like Z-Wave or use or make something like SwitchMote. You can program the SwitchMote yourself to all your wishes.

I have retrieved the code of my remote with an oscilloscope. Here you can see the last portion of the message.



The KaKu remote controls send commands to the sockets via OOK (OnOff Keying) modulation over a 433MHz carrier. The protocol of the KaKu consists in total of 32 bits:

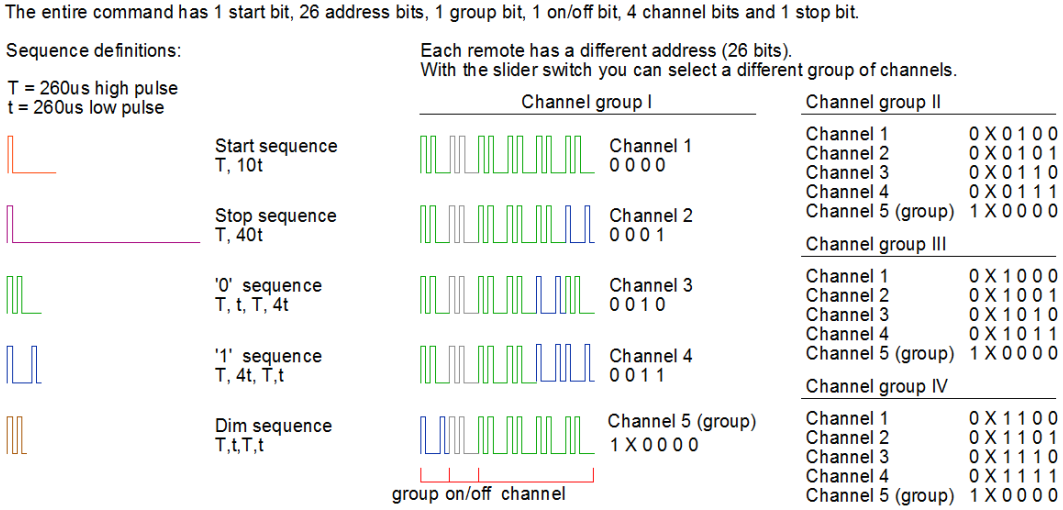
- 1 startbit
- 26 addressbits
- 1 group bit
- 1 on/off bit or dim bit
- 4 channel bits
- 4 optional dim level bits



In case of a dim command, the on/off bit is replaced by the dim bit. In this case there are 4 additional bits between the channe bits and the stop bit to specify the dim level. Because of the 4 bits the dim level has 16 levels. If the dim level is 0 the light will be off, but it is not the same as turning the socket off. In this case the entire command is 36 bits long.

The address is different for each Remote Control. This is the sequence of my Remote Control. It is hard coded in oppose to models with selector switches to select the address. Due to the 26 bits, the chance for 2 remotes with the same address is quite small.

In the picture below the timing is defined. The T stands for the period time and is in the image 260us. I have tested with different values and it still works as long as the period is between 150 and 295us. I have chosen the 260us, because this approaches the signal of the remote control at best.



The entire sequence is repeated 8 times to make sure the command is received.

Teensy 3.1

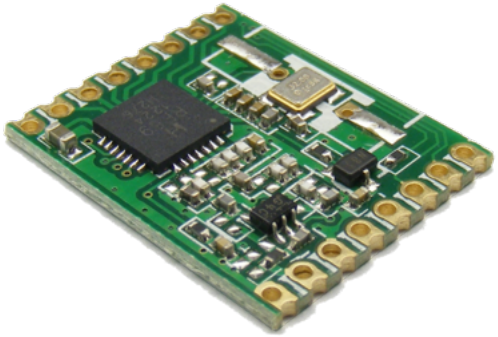
To achieve this I am actually not using an Arduino, but a Teensy 3.1. This is a similar product, but is has better specs and is cheaper (€ 12,30 at OSH Park).

The Teensy has more IO, runs on 72MHz instead of 16MHZ and is 6 times smaller. It has an ARM processor, but you can use the normal Arduino IDE. You do need to install an add-on, but once you have done that, it works just fine. Writing the program into the Teensy is faster too, which is a nice additional advantage. It plugs right into a breadboard, which makes things very easy when you are developing.

RFM69W

Back to t

To send and receive commands I use a RFM69W Rev 2.0 transmitter. These are available in several frequencies, but I use th 433MHz version, the same frequency the KaKu uses. You should specify the requested frequency when you buy the product. have never worked with RF transmitters before and I have no experience with all its characteristics, so keep that in mind whe you read the text below. In the text I refer to a data sheet (RFM69W-V1.3) which can be found here.



The transceiver can handle FSK and ASK modulation and OOK. For this application I need OOK, which stands for On-Off Keying. This basically means that to send a signal you turn the 433MHz radio signal on and off for certain periods of time. Th transceiver has a lot of register settings. These are standard set to use FSK modulation, so it takes quite a bit of tweaking to figure out which settings need to be changed to send en receive OOK messages properly.

Connecting the RFM69W

First we have to connect the RFM to the Teensy. First we have to connect the power and the SPI bus. Now the Digital IO of th RFM. DIO0 is an interrupt signal, which is used to signal the Teensy a message has arrived. The interrupt is generated as soon as a package is available in the FIFO. In OOK mode this signal is not used, but in FSK mode it should be connected to the Teensy interrupt 0, I think that is also pin DI0. I think on the Arduino interrupt 0 is connected to pin 2. Then connect the RFM DIO2 to the Teensy DI15. This is the data pin. If you want to send data, you should put the sequence on DIO2, if you are receiving data, you will receive in on DIO2. So this pin works bi-directional. For testing purposes I have connected RFM DIO3 to DI16. This will give me a signal when data is received (RxReady). And, finally, RFM DIO 5 to DI17. I use this pin to read the RSSI (Received Signal Strength Indicator) signal. This way you can see if the 433MHz radio transmitter is on.
Here is an overview of the pin-out:

Pin on RFM69W	Pin on Teensy	Description
3.3V	3.3V	Power supply: 3.3V
GND	GND	Power supply: Ground
NSS	DI10	SPI bus: Slave Select
MOSI	DI11 (DOUT)	SPI bus: Master Out, Slave In
MISO	DI12 (DIN)	SPI bus: Master In, Slave Out
SCK	DI13 (SCK)	SPI bus: Clock signal
DIO0	DI0	DIO: Interrupt signal from RFM
DIO2	DI15/DO15	DIO: Data pin, works bi-directional
DIO3	DI16	DIO: RxReady signal from RFM
DIO5	DI17	DIO: RSSI signal from RFM
ANA		Antenna

The RFM should also have an antenna. This can be just a simple straight wire, although the length of the wire is very important. This must be 1/4 of the wavelength. The wavelength can be calculated with the following formula:

*c = frequency * wavelength*

In this formula, c is the speed of light, which is 3*10^8m/s. This means the wavelength is:

wavelength = c / frequency

wavelength = 300 000 000 / 433 000 000 = 0.69m

The length of the wire should be 1/4 of the wavelength:

*1/4 * 0.69m = 0.173m = 17.3cm*

This 17.3cm long wire should be soldered on the RFM ANA pin.

Initializing the RFM69W

First we have to initialize the RFM with some basic settings. There are several settings that need to be modified, the rest can stay at the standard value. These standard values can be found in the data sheet.

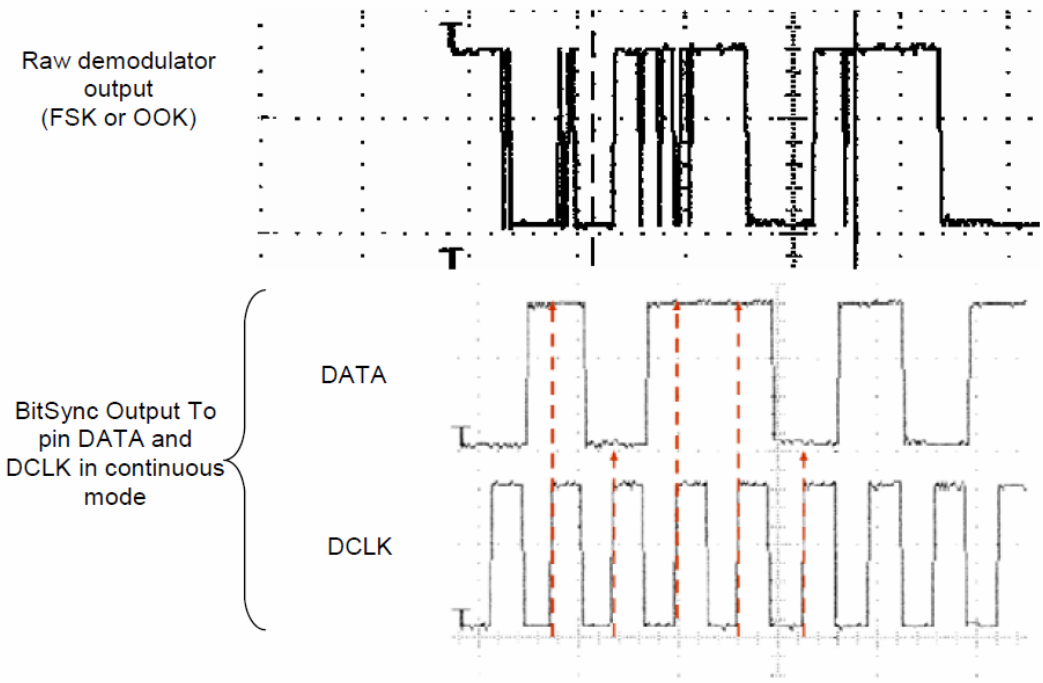
With the first setting we can tell the RFM in which mode it should start. We start in Standby mode with the Sequencer On. When you are switching from one mode to another, the sequencer takes the RFM through all the necessary steps. If you for example are going from Sleep Mode to Transmit Mode, the sequencer will first go to Standby Mode, then to Frequency Synthesizer Mode, and finally, when the PLL has locked, to Transmit Mode. It can be set with the register setting RegOpMode

```
0x01   RegOpMode          0000 0100 (BIN)     0x04 (HEX)
```

Then we set the RFM in OOK mode Without Bit Synchronizer and with No Shaping. I think the Synchronizer is for synchronizing with a clock signal. I have tried using it and it does give a clean signal, but also corrupts the signal which make it useless so far. Maybe if all the settings and filters are correct, you will get a proper signal, but for now I leave it at No Shaping:

```
0x02   RegDataModul       1101 0000 (BIN)     0xD0 (HEX)
```

If you turn the Bit Synchronizer on, a clock is used to synchronize the data on the output. This way you get a clean and synchronized digital signal, free of glitches. The data is available on DIO2, the clock is made available on pin DIO1/DCLK. If you want the clock available on DIO1, make sure it is set properly in 0x26 RegDioMapping2. It is also possible to divide the clock signal on DIO1. The clock speed of FXOSC is 32MHz. The Dio0Mapping in 0x25 RegDioMapping1 should be 00, according to page 48 and 69 of the data sheet. The Bit Synchronizer can be enabled or disabled by changing the DataMode i register setting 0x02 RegDataModul.



The next 2 settings are for the bit rate of the SPI bus. I leave these at the default value of 4.8kb/s:

```
0x03   RegBitrateMsb      0001 1010 (BIN)     0x1A (HEX)
0x04   RegBitrateLsb      0000 1011 (BIN)     0x0B (HEX)
```

Next, we set the frequency of the carrier. For communicating with the KaKu, we need a carrier frequency of 433MHz. The default value is 915MHz, which is in Hex E4 C0 00 and binary:

1110 0100 1100 0000 0000 0000

The value 915 is held in the first 10 bits. So if you remove the last 14 zero's, the remainder is:

1110 0100 11

which is 915 (DEC). If you do this same trick reversed for 433MHz, you will have to convert 433 to binary (10 bits):

0110 1100 01

If you add 14 zero's, you will get:

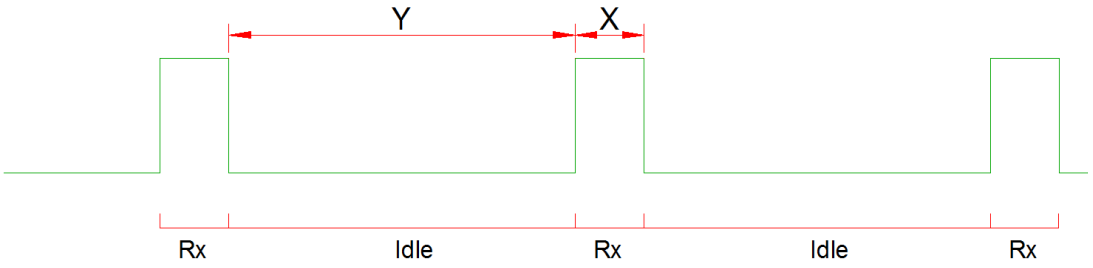
0110 1100 0100 0000 0000 0000

Convert this to Hex, which is 6C 40 00.

```
0x07   RegFrfMsb          0110 1100 (BIN)     0x6C (HEX)
0x08   RegFrfMid          0100 0000 (BIN)     0x40 (HEX)
0x09   RegFrfLsb          0000 0000 (BIN)     0x00 (HEX)
```

Now we arrive at the timing settings in Listen Mode. These settings can be set as stated below, but first I will explain what these settings mean. This is where it gets a bit more complex. When we switch from Standby Mode (84 HEX) to Listen Mode

(C4 HEX) and you would check the mode regularly, you will notice that the RFM most of the time will be in Listen Mode (C4 HEX), but sometimes switches briefly to Receive Mode (D0 HEX). This is shown in the picture below.



In this illustrative picture we can see that the RFM is most of the time in Standby Mode (idle) and than shortly Receiving Mode (Rx). The idle time can be set with two settings, ListenResolIdle (in register 0x0D RegListen1) and ListenCoefIdle (register 0x0E RegListen2). If you multiply these two variables, you will get the idle time (X):

*X = LitenResolIdle * ListenCoefIdle*

Bear in mind the latter value is stored as a hexadecimal value, but for the calculation you will need the decimal value. The receiving time can be set with the settings ListenResolRx (in register 0x0D RegListen1) and ListenCoefRx (register RegListen3). If you multiply these two variables, you will get the time it is receiving (Y):

*Y = ListenResolRx * ListenCoefRx*

In our case, one entire sequence takes about 62ms (with T=260us). If the sequence is repeated 8 times to make sure the message will be received, the total message will be 8 * 62ms = 496ms. So if we check at least every 300ms, we are sure that we receive a portion of the message with at least one sequence in it. So we should have an idle time (X) of approximately 300ms. If we leave LitenResolIdle at the standard value (4.1ms), we can calculate ListenCoefIdle.

ListenCoefIdle = X / LitenResolIdle

ListenCoefIdle = 300 / 4.1 = 73 (DEC).

Convert this to Hexadecimal, we get 49 (HEX).

Now the Receiving time. This period should be longer than the longest 'low' time, which is the 'low' of the stop bit. This is 40 * 260us = 10.4ms, so we should have a Rx time (Y) of approximately 11ms. If we leave ListenResolRx at the standard value (64us), we can calculate ListenCoefRx.

ListenCoefRx = X / ListenResolRx

ListenCoefRx = 11 / 0.064 = 170 (DEC).

Convert this to Hexadecimal, we get AA (HEX). So the entries for the registers would be:

```
0x0D   RegListen1          1001 0010 (BIN)     0x92 (HEX)
0x0E   RegListen2          0100 1001 (BIN)     0x49 (HEX)
0x0F   RegListen3          1010 1010 (BIN)     0xAA (HEX)
```

The antenna is connected to the LNA (Low Noise Amplifier). The recommended value for the LNA's input impedance is 200 Ohms according to the specifications. I read an article of someone who tested the reception with 50 Ohms and 200 Ohms and he had 7-8 dBm better reception with 200 Ohms. The current LNA gain can be read from the variable LnaCurrentGain (read-only). With the LNA gain setting you can decide how much the input signal is amplified. I leave at the standard value of 000 for now, which means that the AGC (Automatic Gain Control) takes care of the optimal gain setting. Maybe I will do some tests later, because if the gain is to high, you might also receive a lot of noise.

```
0x18   RegLna              1000 1000 (BIN)     0x88 (HEX)
```

The bit rate can be calculated with this formula, where T is the period (260us):

BR = 1 / T

BR = 1 / 0.00026 = 3846bps

The Bw can be set to 2 * BR = 7692Hz.
This means according to table 14 of the data sheet RxBwMant can be set to 16 and RxBwExp to 5.

DC cancellation is required in zero-IF architecture transceivers to remove any DC offset generated through self-reception. The standard value is 4% of the RxBw. It would be interesting to try what would happen if you put it at almost zero. I assume you should be able to make it visible on the oscilloscope. Maybe I will try that later, for now I leave it at the standard value of 4% of the RxBw.

```
0x19   RegRxBw             0100 0101 (BIN)     0x45 (HEX)
```

The recommended mode of operation is the Peak threshold mode. This means the threshold does not stay the same all the time, but when the RSSI Receiving Signal Strength Indicator) changes, the threshold value changes too. The threshold level starts at floor level, defined in OokFixedThresh. When a signal is received, the threshold values rises until 6dB under the RSSI signal strength. When the signal is gone, the threshold drops gradually according to the settings of OokPeakThreshStep and OokPeakThreshDec. When a signal drop can be expected these values can be altered to prevent data loss.
I suppose 'chip' in OokPeakThreshDec refers to the chip rate. In our case, a '0' or '1' consists of 7 chips, one chip is 260us. The chip rate is the amount of chips per second.
How the value of OokFixedThresh should be determined, is described in figure 12 of the data sheet. It puts the floor threshold on a level where no noise is received, so this is an important setting. The standard value is 6dB.
I assume OokAverageThreshFilt (0x1C) is some kind of filter to get rid of spikes. I leave it at it's original value.

```
0x1B   RegOokPeak          0100 0000 (BIN)     0x40 (HEX)
```

```
0x1D   RegFixedThresh      0000 0110 (BIN)     0x06 (HEX)
```

In the registers RegDioMapping1 en RegDioMapping2 we can configure the IO. I use DIO2 for reading reading and writing th data. If you want to send something, you have to offer the pulse train to DIO2. If you receive data, you receive it on DIO2. Thi pin works bi-direcitonal.
DIO3 is triggered when the data is received (RxReady).
DIO5 is used to make the RSSI signal visible on the scope. It is not actually needed in the Teensy.
The clock signal could also be read on DIO5, but I don't need the clock.

```
0x25   RegDioMapping1      0000 0001 (BIN)     0x01 (HEX)
0x26   RegDioMapping2      0001 0000 (BIN)     0x10 (HEX)
```

The RssiThreshold is default 0xE4, which can be converted do dBm with the formula:

- RssiThreshold / 2

So if we apply this to the default value 0xE4, which is 228 (DEC), we get:

- 228 / 2 = -114 dBm

If I set the value to 228, the RFM is receiving something immediately, even when I am not sending something. This means it i receiving humbug. If I set the value to 160, it is often not responding. I have to send a message (push the button on the remo control) several times before it is receiving. If I set the value to 180 (B4 HEX), it seems to be responding well, also from a larger distance. I am not sure yes what the difference is between the RSSI Threshold and the OokFixedThresh.
The current RSSI level can be read by reading 0x24 RegRssiValue (read only).

There is also a Timeout signal available. When you are listening for a certain time but not receiving a valid message, the listen mode will be timed out and the RFM will go back to Standby mode. For now, the Timeout is switched off.

```
0x29   RegRssiThresh       1110 0100 (BIN)     0xB4 (HEX)
0x2A   RegRxTimeout1       0000 0000 (BIN)     0x00 (HEX)
```

This was the last setting for initializing the RFM in OOK mode. There is no need look at the Packet Engine Registers, those are used in the FSK mode.

Transmitting with the RFM69W

Sending a message is very simple. All you have to do is turn the radio on and offer the data (the pulses) to DIO2. When you are done, turn off the radio and go back to Listen Mode or Sleep Mode, depending on what you want to do. Make sure you send one message several times (8 times) to make sure the message will be received. You can turn the radio on by putting it in Transmit mode by changing the RegOpMode:

```
0x01   RegOpMode           0000 1100 (BIN)     0x0C (HEX)
```

Now the data can be given from the Teensy output DO15 to RFM DIO2. Make sure pin 15 on the Teensy is working as an output pin.
When the message is sent, you can go back to Receiver mode by changing the RegOpMode to:

```
0x01   RegOpMode           0001 0000 (BIN)     0x10 (HEX)
```

Receiving with the RFM69W

If you want to receive a message, make sure you are in Receiver Mode:
When the message is sent, go back to Receiver mode by changing the RegOpMode to:

```
0x01   RegOpMode           0001 0000 (BIN)     0x10 (HEX)
```

When you would be reading the mode from the serial monitor with the Arduino software, you would notice the RFM is in Standby Mode most of the time and will switch to Receiving mode shortly every now and then. The timing of this process is described above (0x0D, 0x0E, 0x0F).

Arduino sketches

To use the Teensy with Arduino software, make sure you install the Teensyduino add-on. The Arduino boards have Atmel chips, the Teensy uses an Arm chip. The software can be downloaded from www.prjc.com.

Libraries

To control the RFM in a way it can communicate with the KaKu modules, we will need some libraries. First we need library for the SPI bus. I think the SPI library is already supplied with the Arduino software.

Secondly we need a library send and receive KaKu commands. There already is a library for this. You can find in if you look f NewRemoteSwitch, available on BitBucket. This library does not use a RFM69W, but a separate transmitter and receiver. Each can be controlled by a single data pin, which makes them very easy to use. Since I am using the RFM69W, I have to alter the library. The altered library is available here.

There is also a library available on github to control the RFM69W. This library does not support OOK mode, only FSK mode. This means we have to adjust this library to our needs. I added an initialisation function for OOK mode. I also added a functio to turn the radio on and off easily.

Arduino sketch

In the Arduino sketch we need to include the RFM69W library and the KaKu library.

sample.ino

```
/*
  Blink
  Turns on an LED on for one second, then off for one second, repeatedly.

  This example code is in the public domain.
 */

// Pin 13 has an LED connected on most Arduino boards.
// give it a name:
int led = 13;

// the setup routine runs once when you press reset:
void setup() {
  // initialize the digital pin as an output.
  pinMode(led, OUTPUT);
}

// the loop routine runs over and over again forever:
void loop() {
  digitalWrite(led, HIGH);   // turn the LED on (HIGH is the voltage level)
  delay(1000);               // wait for a second
  digitalWrite(led, LOW);    // turn the LED off by making the voltage LOW
  delay(1000);               // wait for a second
  Serial.print("Hilco");
}
```