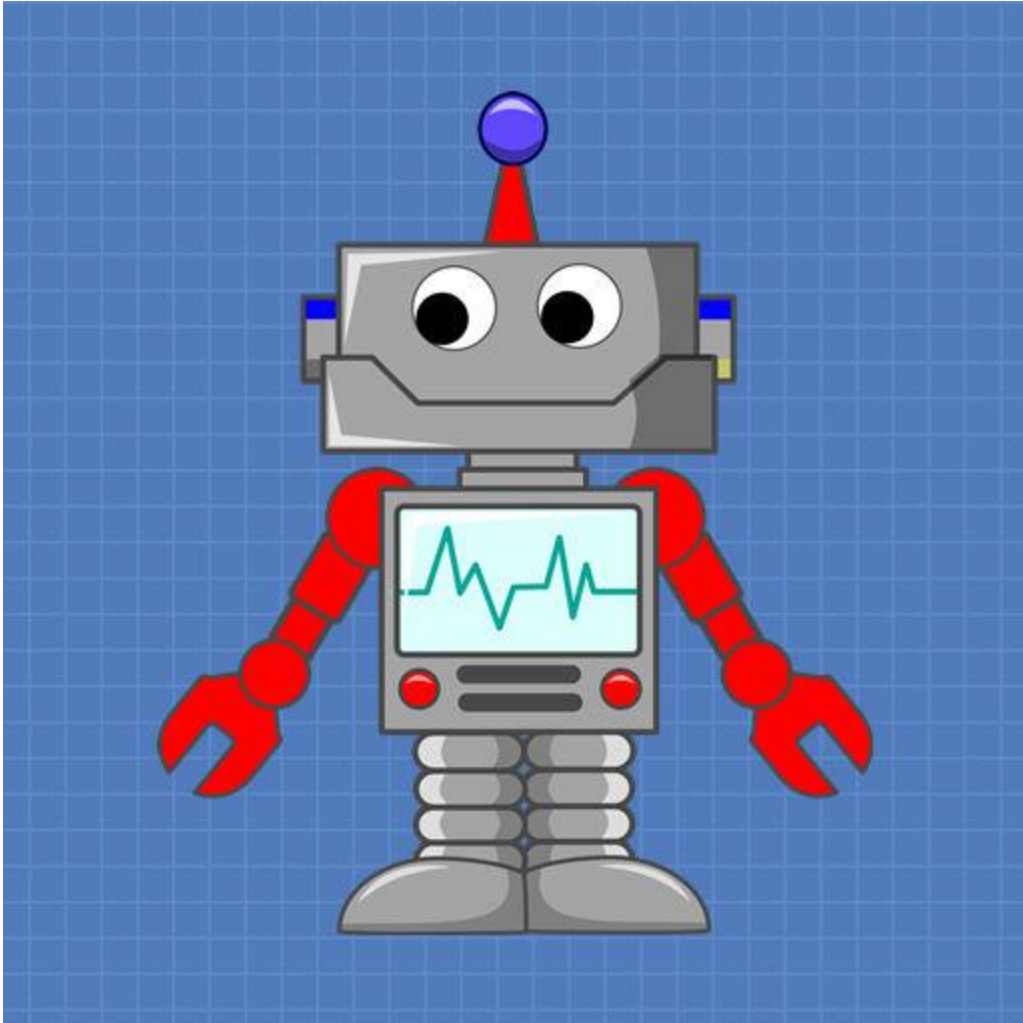


IR Remotes Revisited - 2023



DroneBot Workshop Tutorial

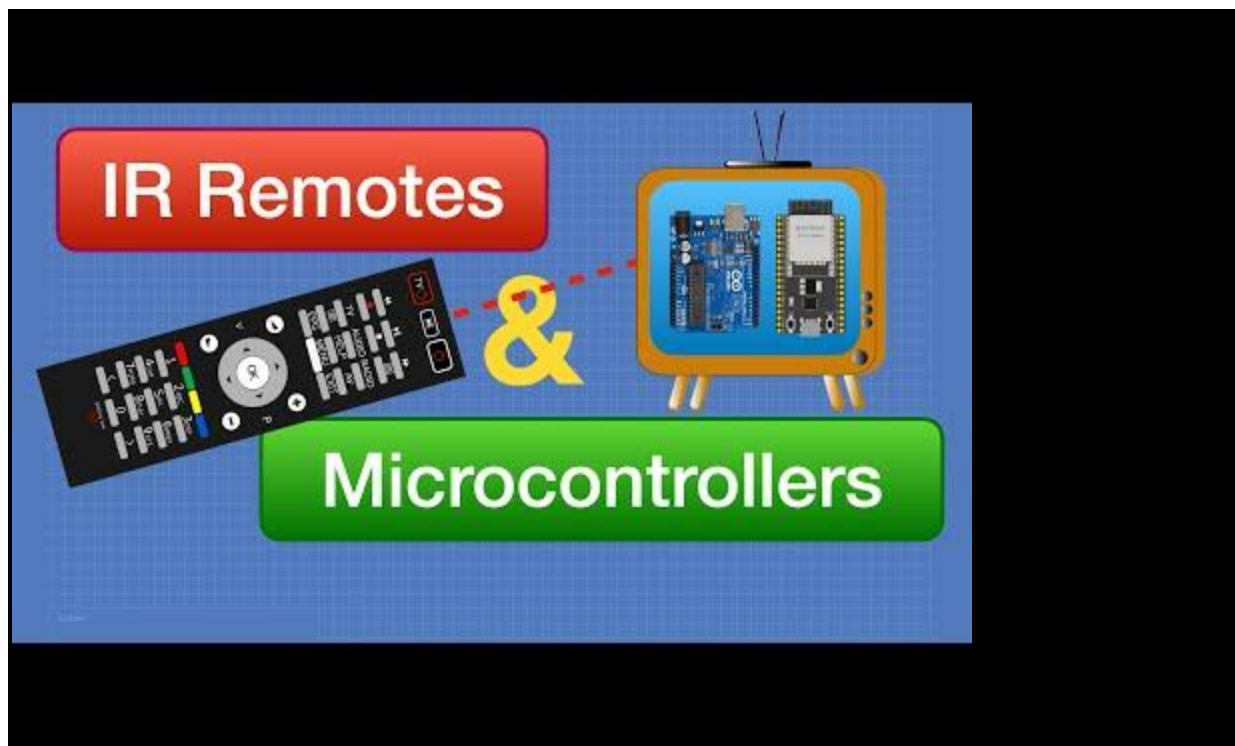
<https://dronebotworkshop.com>

For more projects and tutorials visit the DroneBot Workshop - <https://dronebotworkshop.com>

After five years, it's time we took another look at using infrared remote controls with microcontrollers. We'll work with some updated libraries and see what we can do to decode those "stubborn" air conditioner remote controls. We'll also take a closer look at the hardware used to exchange IR signals. And, of course, we'll build many fun IR remote projects!

Introduction

It's a safe bet that you own at least one infrared remote control. In fact, it's likely that you own several, perhaps so many that you would have trouble accounting for them all. Once exclusively used for televisions, remote controls are now included with all varieties of electronic gear.



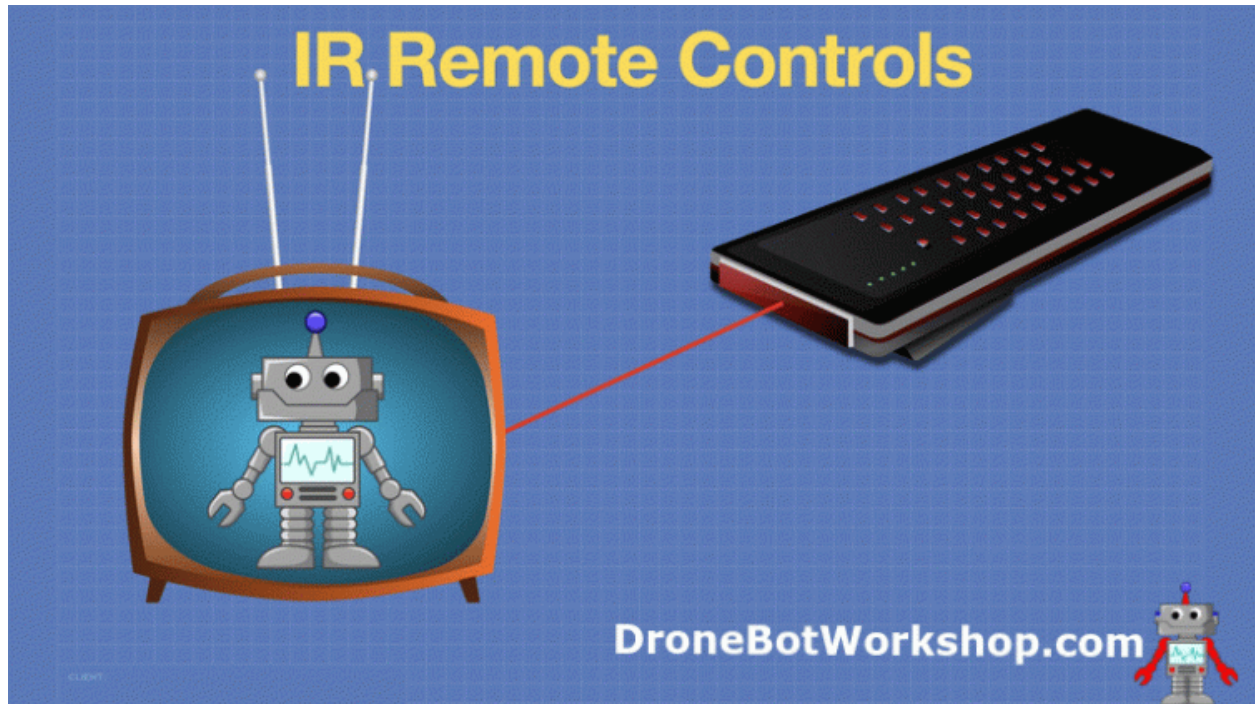
IR Remote controls offer numerous possibilities for experimenters and microcontroller enthusiasts. We can use microcontrollers to emulate remote controls, placing a wide

<https://dronebotworkshop.com>

For more projects and tutorials visit the DroneBot Workshop - <https://dronebotworkshop.com>

variety of devices under the control of our programs. We can also get our micros to respond to IR remote controls, providing a convenient way to control our projects.

Microcontrollers also open up several advanced capabilities, such as converting remote control codes from one manufacturer to another or playing back sequences of remote control codes.



Today, we will see how these little gems work and how we can use them in our projects.

IR Remote Controls

Televisions, air conditioners, sound systems, electronic blinds, humidifiers – all of these appliances come equipped with infrared remote controls. Usually, they play nice together, although occasionally, we get into a conflict, and turning up the sound on the TV also changes the humidifier settings.

<https://dronebotworkshop.com>

For more projects and tutorials visit the DroneBot Workshop - <https://dronebotworkshop.com>

Let's learn more about these devices to see if we can understand how they work, how we can use them in our projects, and why we have so many of these critters on our coffee table!

History of Remote Controls

It's hard to imagine life without remote controls, but believe it or not, there was once a time when you actually had to get out of your chair to change channels or adjust the sound on the TV!



The remote control has a rich history that spans several decades. The first television remote control, named “*Lazy Bones*,” was developed by Zenith Radio Corporation in the late 1940s. However, it was connected to the TV with a wire, and at 30 dollars (387 dollars in 2023), it was an expensive accessory.

For more projects and tutorials visit the DroneBot Workshop - <https://dronebotworkshop.com>

[illegible]

In 1955, Zenith's Eugene Polley developed the first wireless remote control called the "*Flashmatic*." It worked by directing a visible light beam toward photocells on the television set. As you might suspect, this design was prone to interference from other light sources, so it wasn't very successful.

<https://dronebotworkshop.com>



For more projects and tutorials visit the DroneBot Workshop - <https://dronebotworkshop.com>

[illegible]

As consumer electronics proliferated, so did the applications of IR remotes. They started being used for VCRs, DVD players, stereos, and eventually, air conditioners, cameras, and more. These days, they are everywhere.

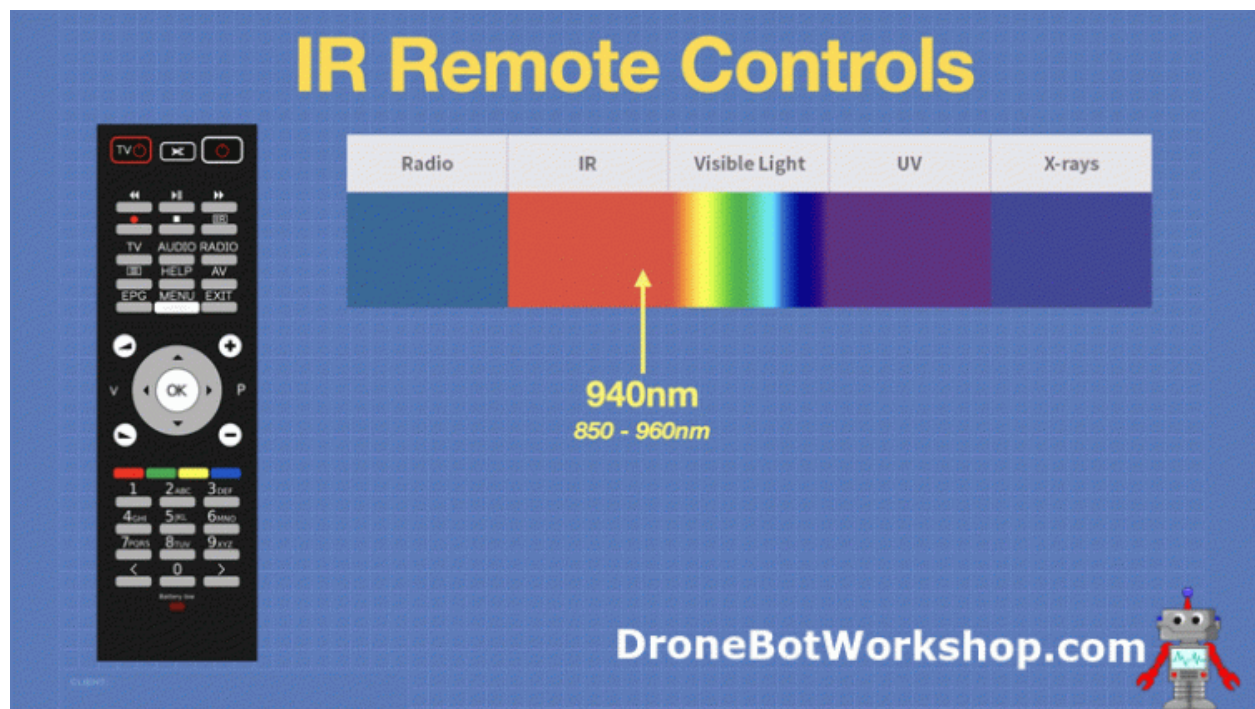
<https://dronebotworkshop.com>

Infrared Remote Control Basics

In its most basic form, an IR remote control sends a pulsed beam of infrared light, which contains the key code for the button you pressed. The receiver extracts the code and performs the desired operation.

Infrared Light

The light these devices use is in the “infrared” range, meaning the range of light below red in the visible spectrum.



Light can be measured by its wavelength, and most infrared remotes operate in wavelengths ranging from 850 to 960 nanometers. By far, **the most common wavelength is 940nm.**

Carrier Frequency

The pulses of infrared light are also modulated with a carrier wave, just like a radio signal. This technique allows the receiver to distinguish the remote signal from the infrared background noise, which can sometimes be quite high.

Carrier frequencies can vary between 30 and 60 kilohertz, but **the most common carrier frequency is 38 kHz**.

Modulation Techniques

All remote control data is sent in a serial format. The data is encoded using a number of different modulation techniques. These different techniques determine how the zeros and ones are formatted for transmission over the infrared light beam.

There are three basic modulation techniques used, along with several variations of them.

- **Pulse Density Modulation (PDM)** – Also called *Space Encoding* or *Pulse Distance Modulation*.
- **Pulse Width Modulation (PWM)** – Also called *Pulse Encoding*.
- **Bi-Phase Modulation** – Also called *Manchester Encoding*.

The following illustrations show how the data is formatted using these three modulation techniques:

IR Remote Controls


PDM

PWM


Bi-Phase

Pulse Density Modulation

- Also called **Space Encoding** or **Pulse Distance Modulation**
- Modulates the **duration** of the space **between** pulses
- **0 = Short Space, 1 = Long Space**



DroneBotWorkshop.com



IR Remote Controls


PDM

PWM


Bi-Phase

Pulse Width Modulation

- Also called **Pulse Encoding**
- Modulates the **width** of the pulses
- **0 = Narrow Pulse, 1 = Wide Pulse**



DroneBotWorkshop.com



IR Remote Controls


PDM

PWM

Bi-Phase

Bi-Phase Modulation

- Also called **Manchester Encoding**
- Modulates the **transition** between high and low levels
- **0 = High to Low** transition, **1 = Low to High** transition



DroneBotWorkshop.com

IR Remote Protocols

Consider those remotes that are taking over your coffee table. Each of them has numerous functions, yet they seldom interfere with one another. This may be due to different IR wavelengths or carrier frequencies, but 940nm and 38kHz are pretty well the standard. It's likely every remote you own uses these standards.

So, how do they keep from interfering? One way is by using different protocols.

IR remote protocols define all the parameters for exchanging IR data. Some of these parameters include:

- The modulation technique used.
- The timing and format of the start bit.
- The number of address and data bits.
- The format of any error checking used.
- The format of the EOM (End Of Message) or stop bit(s).


- Whether the data is LSB (Least Significant Bit) or MSB (Most Significant Bit) first.

There are many different protocols in use, common ones as well as propriety ones. The chart below shows only a small sampling of protocols in common use; there are actually hundreds of them.

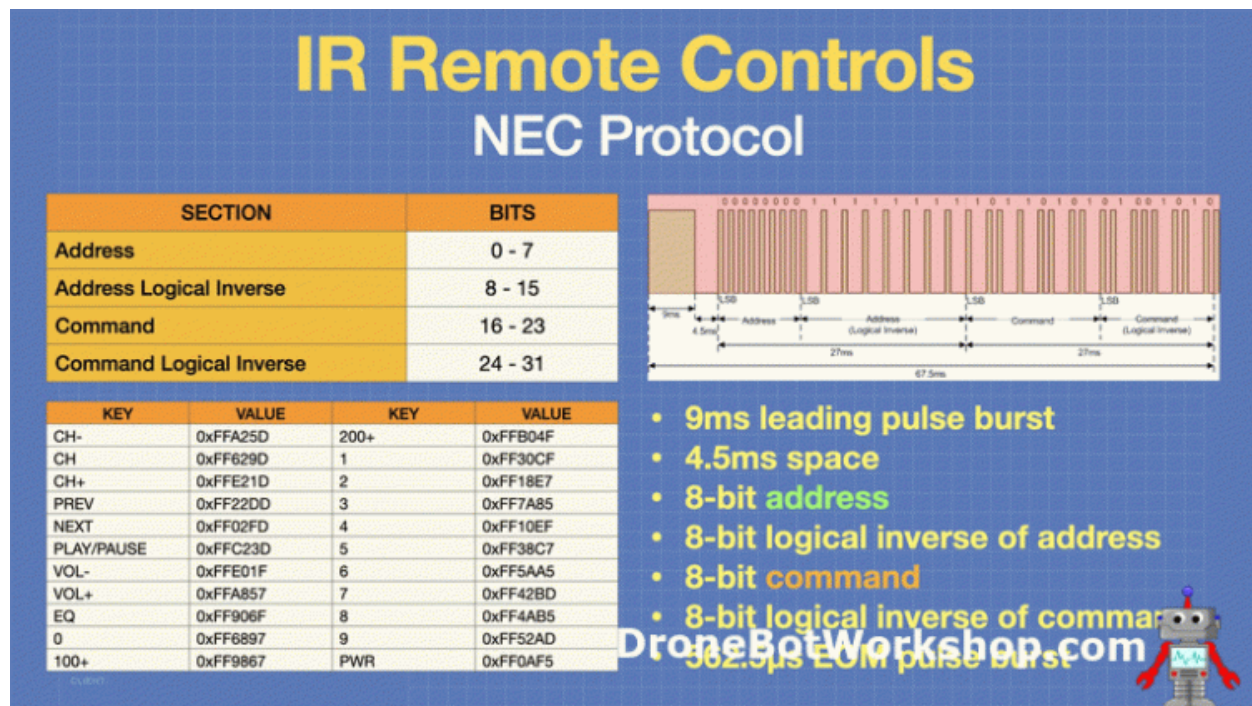
NEC IR Remote Protocol

The most popular infrared protocol is the NEC protocol. Hundreds of manufacturers use it, and it is easy to work with as it is extensively documented. It's likely used on most of the remotes you own, except for the "big brand name" ones like Sony, LG, Panasonic, Apple, and Samsung, who use their propriety protocols.

IR Remote Controls			
PROTOCOL	DESCRIPTION	BITS	MODULATION
NEC	Most popular protocol used in consumer electronics	32	PDM
RC-5	Developed by Phillips, no longer used	14	Bi-Phase
RC-6	Sucessor of RC-5, supports more features	20, 36, 44	Bi-Phase
Sony SIRC	Sony Infrared Remote Control	12, 15, 20	PDM or Bi-Phase
Sharp	Used in most Sharp products	12	PDM
Panasonic	Developed by Panasonic, primarily for televisions	22	PDM
JVC	Developed by JVC, primarily for audio equipment	12, 24	PDM
Samsung	Used in most Samsung products	20	PDM
Logitech	Logitech Harmony remote protocol	Variable	PWM
LG	Used in most LG products	28	PDM
Denon	Used in most Denon products	16	PDM
XMP	Extended Mode Protocol, customizable packet structure	variable	PDM or PWM



This chart illustrates the structure of the NEC IR protocol, as well as a few example commands. Note that the packet begins with a 9ms leading pulse burst followed by a 4.5ms space.



This is followed by 32 bits of address and command data. Note that both the address and command are 8 bits, and for each, a logical inversion is also sent. The logical inversion serves as a form of error checking.

For our purposes, the only data we need to work with are the Address and Command values. This is true for other protocols, as well as the NEC one.



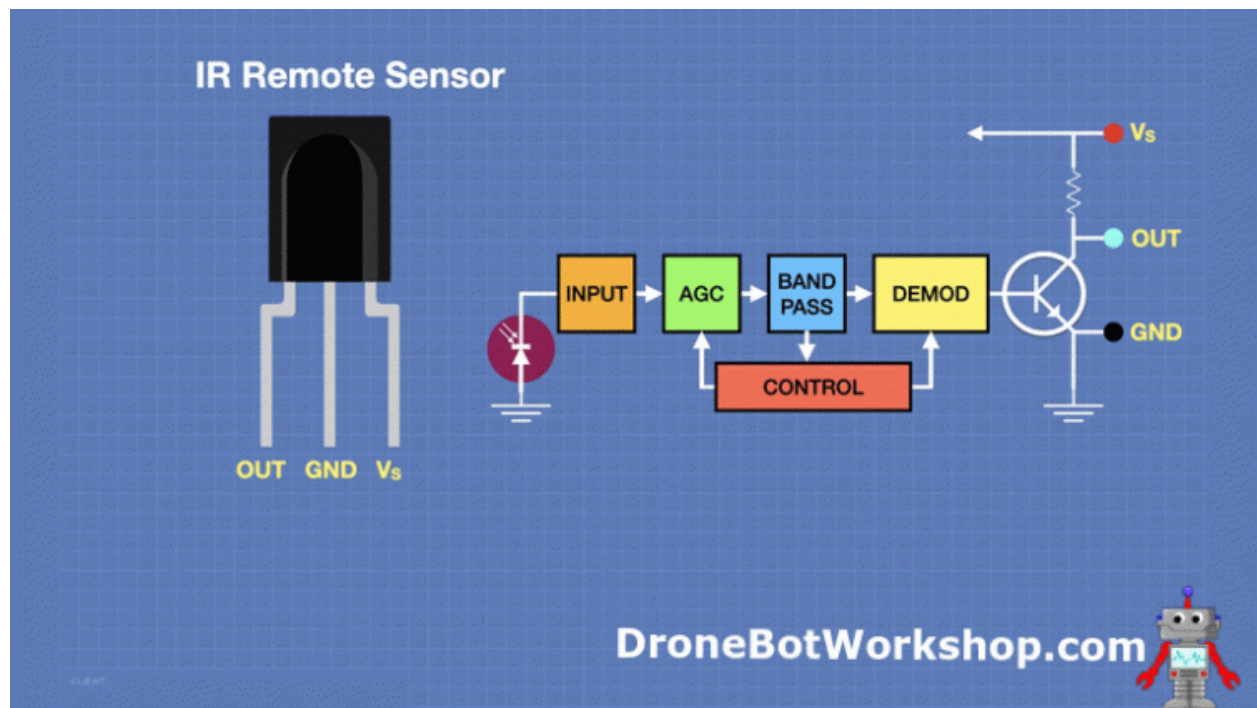
In this illustration, we can see that using a different address allows two remotes that use the NEC protocol to coexist in the same room without interfering with one another. Although both devices use code 0x0A for power, they use different addresses.

IR Remote Control Components

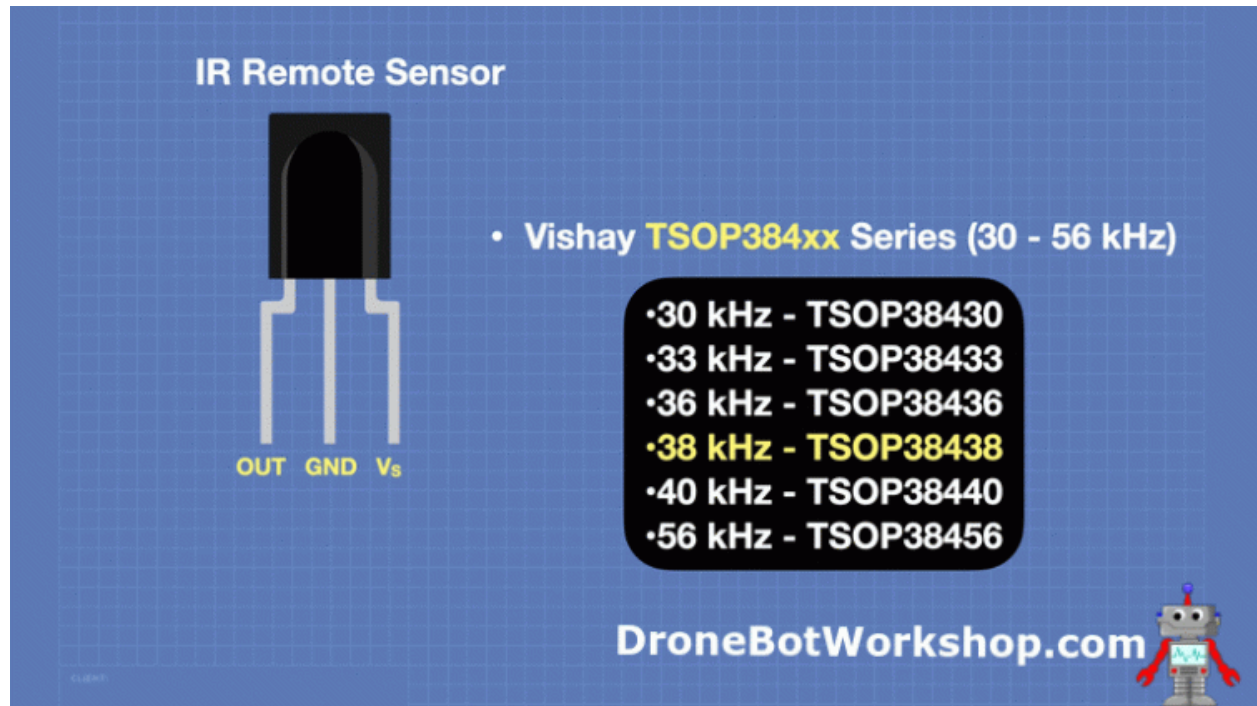
Working with infrared light remote controls requires special components. These are inexpensive and easy to obtain.

IR Receiver Modules

The IR Receiver module packs a lot of functionality into a small package, as shown below:



These modules are available in different IR wavelengths, with 940nm being the most popular. The wavelength is usually keyed into the part number, as shown here for the popular Vishay TSOP34xx series.



TL1838 Sensors

The TL1838 and many other sensors with “1838” in the part number are probably the most common IR sensors available to hobbyists. These devices have been around for decades, and while they certainly work, they lack a lot of the functionality that more modern sensors have.

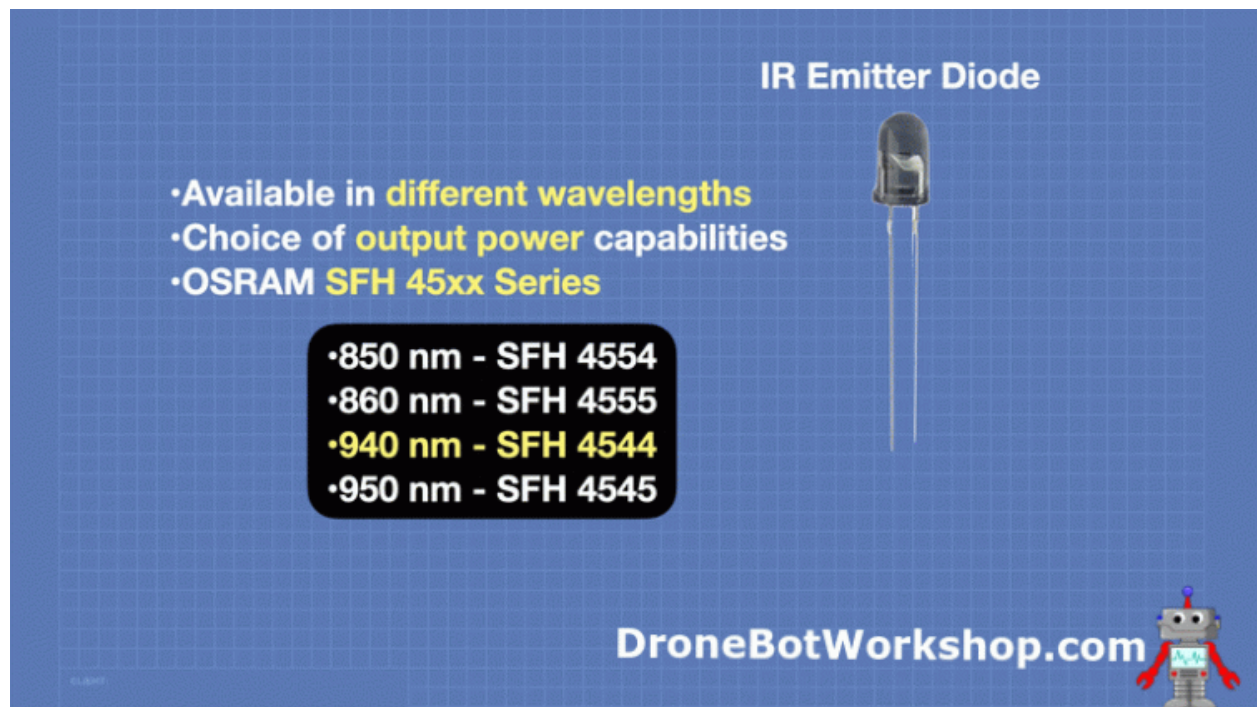
In addition, the TL1838 is meant for 5-volt operation, so it is difficult to use with modern 3.3-volt microcontrollers.

Now, if you pick up one of those packages on Amazon or eBay with a handful of IR emitter diodes and unmarked IR sensors, you likely just picked up a bunch of TL1838 sensors. And while you can certainly use them in your experiments, I would not recommend them for new designs.

IR Emitters

An IR Emitter is essentially an LED that emits infrared light. As with infrared receivers, these are available in different wavelengths.

The illustration below shows the part numbers for the OSRAM SFH 45xx series of infrared emitters, with the popular 940nm model highlighted.



The infographic features a blue background with a grid pattern. At the top right, the title "IR Emitter Diode" is displayed in white. To the left of a small image of an IR emitter diode, there is a list of features in yellow and white text: "•Available in different wavelengths", "•Choice of output power capabilities", and "•OSRAM SFH 45xx Series". Below this, a black rounded rectangle contains a list of part numbers in white and yellow text: "•850 nm - SFH 4554", "•860 nm - SFH 4555", "•940 nm - SFH 4544" (highlighted in yellow), and "•950 nm - SFH 4545". At the bottom right, the website "DroneBotWorkshop.com" is written in white, accompanied by a small robot icon.

IR Emitter Diode

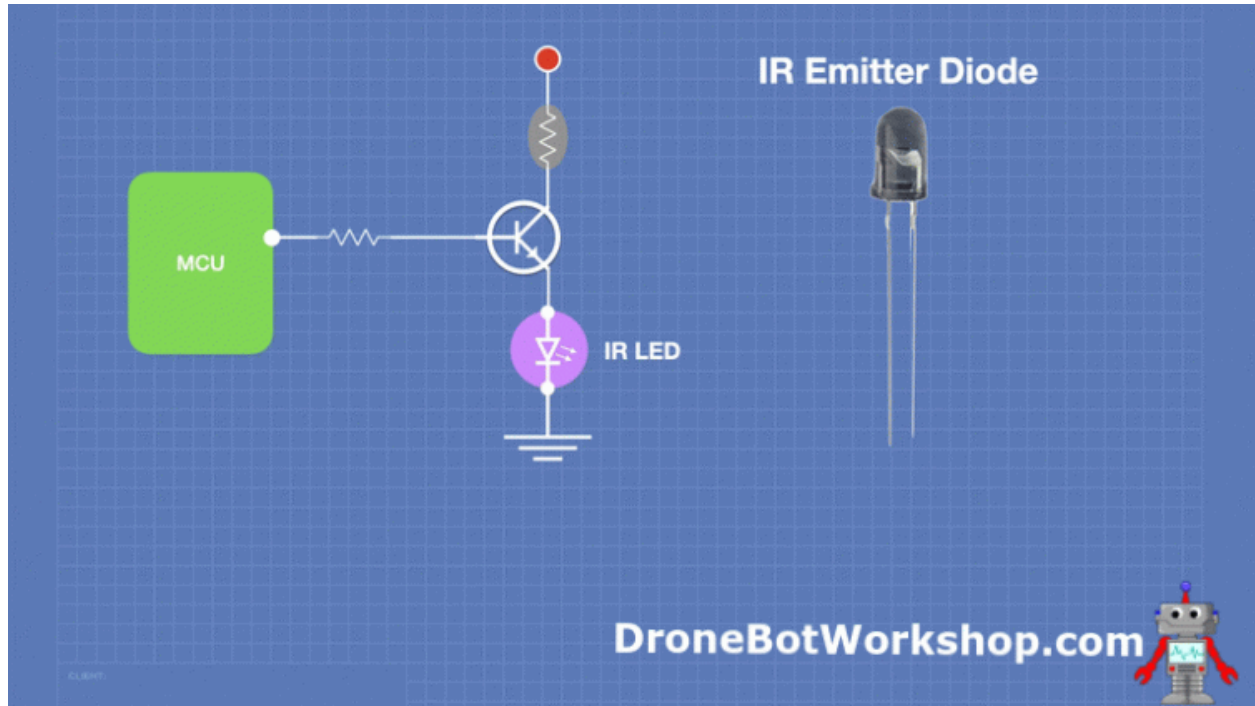
- Available in different wavelengths
- Choice of output power capabilities
- OSRAM SFH 45xx Series

- 850 nm - SFH 4554
- 860 nm - SFH 4555
- 940 nm - SFH 4544
- 950 nm - SFH 4545

DroneBotWorkshop.com

Driving IR Emitters

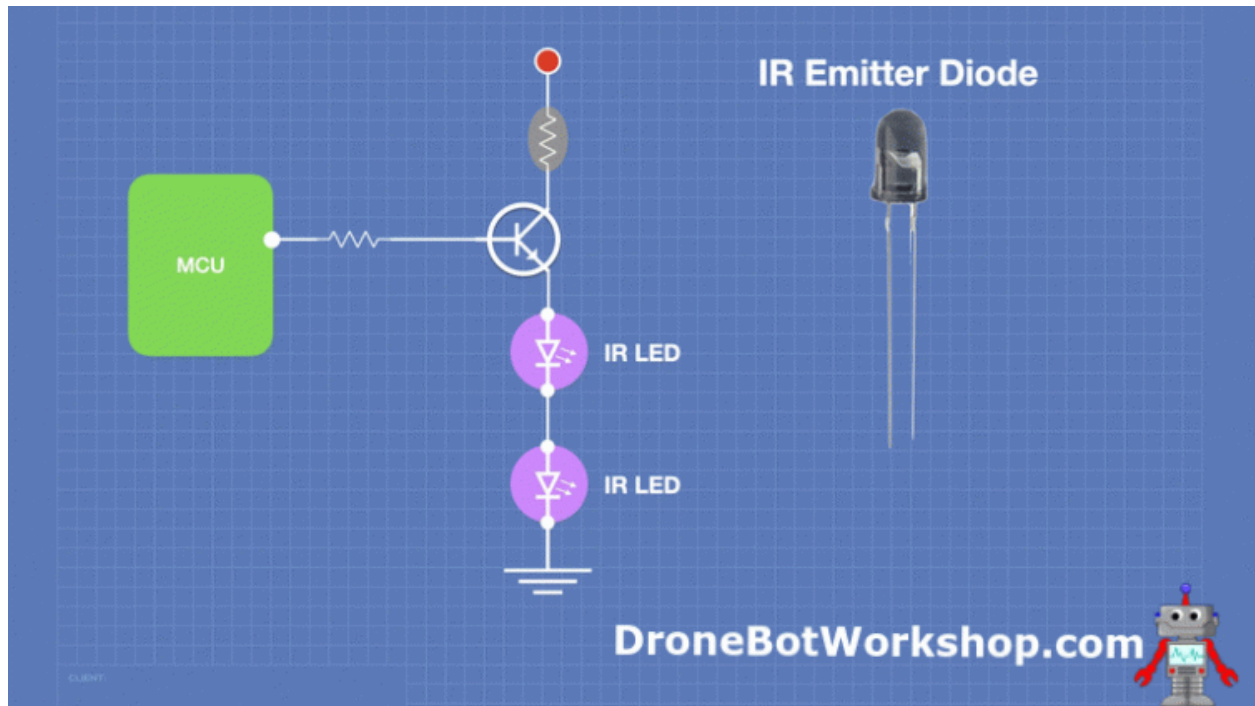
Infrared emitters can consume much more current than a standard visible LED, so driving them directly from a microcontroller's GPIO pin isn't practical. Many of these devices can consume 100ma or more, with some devices allowing up to 1 ampere when pulsed.



In this illustration, a bipolar transistor is used to switch the IR emitter LED. As bipolar transistors are current-driven, a limiting resistor is used on the connection from the GPIO pin to the base of the transistor.

There is also a current limiting resistor for the IR LED. This is usually a low-value resistor; sometimes, a resistor is not even necessary.

For more projects and tutorials visit the DroneBot Workshop - <https://dronebotworkshop.com>

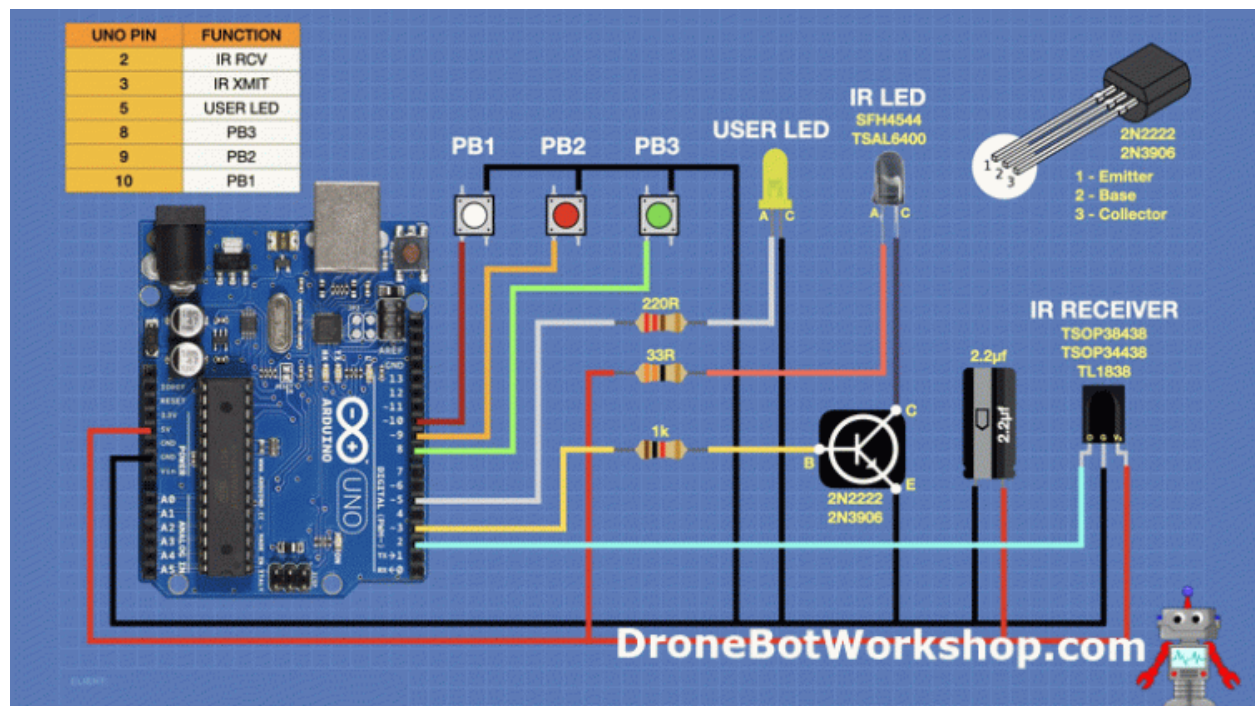


You can also use a transistor to switch multiple IR emitters for increased range and performance.

<https://dronebotworkshop.com>

Arduino Hookup

Here's how we will hook up our Arduino Uno, IR emitter diode, and IR sensor. We will also add three pushbutton switches and a visible LED as well.



Here are a few things to note about this hookup:

- The transistor used to drive the IR emitter is a 2N2222 NPN. You could substitute another NPN transistor if you don't have a 2N2222; make sure you check out the pinout of any substitutions, as they may not match the 2N2222.
- The 2.2uF capacitor is a power supply filter capacitor for the IR receiver module. It should be placed near the sensor's power connection on your breadboard. The value is not critical, and any electrolytic capacitor from 1uF to 10uF can be substituted.
- The 33-ohm dropping resistor for the IR emitter may need adjustment for different IR emitter diodes. In some cases, it may not even be necessary.

For more projects and tutorials visit the DroneBot Workshop - <https://dronebotworkshop.com>

Of course, another thing you'll need, aside from the Arduino hookup and your computer, is a few remote controls to experiment with. You'll need to round a few up; the more, the merrier!

Updated Arduino-IRremote Library

The *Arduino-IRremote* Library is the most popular library for using IR remote controls with microcontrollers. It is constantly being updated, and it was one of those previous updates that made most of the code in the older article obsolete, as the library is now called using another method.

We will install the library in our Arduino IDE and then perform a number of experiments using it.

The easiest way of adding the Arduino-IRremote Library to the Arduino IDE is to install it directly from the library manager.

Once the library is installed, you can access several useful and interesting example sketches.

Arduino-IRremote Library Example Code

There are no less than 25 example sketches included with the Arduino-IRremote Library. Each one is well documented, and the Arduino we just set up is hooked up using the default pins for most of the examples.

The exception is Pin D5, which is referred to as the APPLICATION_PIN in the examples. It is unused in many examples, but in some examples, it is an output, whereas in others, it is a switch input. Currently, I have it hooked up to an LED. I used

For more projects and tutorials visit the DroneBot Workshop - <https://dronebotworkshop.com>

pin D5 as it is also PWM capable, a feature that we will need later on. It can also be used with examples like the *ControlRelay* example instead of the output relay module.

PinDefinitionsAndMore File

Almost all of the 25 IR remote examples use a file named *PinDefinitionsAndMore.h*. As its rather long name would imply, this file contains pin definitions for a variety of microcontrollers, as well as the definitions of some common constants.

The file starts with a list of common microcontrollers and the default pins used. By wiring your microcontroller as per the list, you'll make coding simpler. You can, of course, choose to ignore the default wiring and change the pin definitions in the code, but it's a lot simpler just to follow the list.

Many of the constants we will encounter in the example sketches that follow are contained in the *PinDefinitionsAndMore* file.

SimpleReceiver Example Code

The *SimpleReceiver* is the first example sketch that we will run, and I'm pretty sure you can guess what it does! You'll find it, and all of the other example sketches, in the *File/Examples/Examples from Custom Libraries/IRremote* folder.

```
1 /*
2  * SimpleReceiver.cpp
3  *
4  * Demonstrates receiving NEC IR codes with IRremote
5  *
6  * This file is part of Arduino-IRremote
7  * https://github.com/Arduino-IRremote/Arduino-IRremote.
8  *
9  * *****
10 * MIT License
11 *
12 * Copyright (c) 2020-2023 Armin Joachimsmeier
13 *
14 * Permission is hereby granted, free of charge, to any person obtaining a copy
15 * of this software and associated documentation files (the "Software"), to deal
16 * in the Software without restriction, including without limitation the rights
17 * to use, copy, modify, merge, publish, distribute, sublicense, and/or sell
18 * copies of the Software, and to permit persons to whom the Software is furnished
19 * to do so, subject to the following conditions:
20 *
21 * The above copyright notice and this permission notice shall be included in all
22 * copies or substantial portions of the Software.
23 *
24 * THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
25 IMPLIED,
26 * INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A
27 * PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT
28 * HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION
29 OF
```



```
29 * CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE
SOFTWARE
30
31 * OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
```

```
57 // #define DECODE_DISTANCE_WIDTH // Universal decoder for pulse distance width
58 protocols
59 // #define DECODE_HASH // special decoder for all protocols
60
61 // #define DECODE_BEO // This protocol must always be enabled manually, i.e.
62 it is NOT enabled if no protocol is defined. It prevents decoding of SONY!
63
64 // #define DEBUG // Activate this for lots of lovely debug output from the
65 decoders.
66
67 // #define RAW_BUFFER_LENGTH 180 // Default is 112 if DECODE_MAGIQUEST is enabled,
68 otherwise 100.
69
70 #include <Arduino.h>
71
72 /*
73 * This include defines the actual pin number for pins like IR_RECEIVE_PIN,
74 IR_SEND_PIN for many different boards and architectures
75 */
76
77 #include "PinDefinitionsAndMore.h"
78
79 #include <IRremote.hpp> // include the library
80
81 void setup() {
82     Serial.begin(115200);
83
84     // Just to know which program is running on my Arduino
85
86     Serial.println(F("START " __FILE__ " from " __DATE__ "\r\nUsing library version
87 " VERSION_IRREMOTE));
88
89     // Start the receiver and if not 3. parameter specified, take LED_BUILTIN pin
90 from the internal boards definition as default feedback LED
91
92     IrReceiver.begin(IR_RECEIVE_PIN, ENABLE_LED_FEEDBACK);
```

```
85
86   Serial.print(F("Ready to receive IR signals of protocols: "));
87   printActiveIRProtocols(&Serial);
88   Serial.println(F("at pin " STR(IR_RECEIVE_PIN)));
89 }
90
91 void loop() {
92     /*
93      * Check if received data is available and if yes, try to decode it.
94      * Decoded result is in the IrReceiver.decodedIRData structure.
95      *
96      * E.g. command is in IrReceiver.decodedIRData.command
97      * address is in command is in IrReceiver.decodedIRData.address
98      * and up to 32 bit raw data in IrReceiver.decodedIRData.decodedRawData
99      */
10    if (IrReceiver.decode()) {
10    0
10    1        /*
10    2        * Print a short summary of received data
10    3        */
10    4        IrReceiver.printIRResultShort(&Serial);
10    5        IrReceiver.printIRSendUsage(&Serial);
10    6        if (IrReceiver.decodedIRData.protocol == UNKNOWN) {
10    7            Serial.println(F("Received noise or an unknown (or not yet enabled)
10    8 protocol"));
10    9            // We have an unknown protocol here, print more info
10   10            IrReceiver.printIRResultRawFormatted(&Serial, true);
10   11        }
10   12        Serial.println();
10   13    }
```

```
10
8
/*
10
9    * !!!Important!!! Enable receiving of the next value,
11
11    * since receiving has stopped after the end of the current received data
0 packet.
11
11    */
1
11    IrReceiver.resume(); // Enable receiving of the next value
11
2
/*
11
3    * Finally, check the received data and perform actions according to the
received command
11
4    */
11
11    if (IrReceiver.decodedIRData.command == 0x10) {
5
11        // do something
11
6        } else if (IrReceiver.decodedIRData.command == 0x11) {
11
11        // do something else
7
11        }
11
11    }
8
}
11
9
12
0
12
1
12
2
12
3
```

The sketch starts with definitions for many different remote protocols, with most of them remarked out (i.e., inactive). Only the NEC protocol is enabled by default.

For more projects and tutorials visit the DroneBot Workshop - <https://dronebotworkshop.com>

We include both the library and the *PinDefinitionsAndMore* file, and then move into the Setup, as the pin definitions file has already defined all the constants we need.

In Setup, we start the serial monitor. After that, this sketch (and most of the others) does a print of the current IR Remote version.

We then enable the IR Receiver with this line:

```
1 IrReceiver.begin(IR_RECEIVE_PIN, ENABLE_LED_FEEDBACK);
```

The IR_RECEIVE_PIN is defined as 2 by the pin definitions file and is correct for our Arduino hookup.

The ENABLE_LED_FEEDBACK is a boolean that is set to true. When enabled, the onboard LED will flicker whenever data is received.

Now, on to the Loop:

We start by looking for a decoded output, which will occur whenever a valid IR packet is received.

Once we receive the packet, we print out its contents to the serial monitor. This will include the address, command value, and protocol type. Note that in its default configuration, it will only decode NEC packets; you need to enable additional ones as you require them.

An important thing to note is that the receiver stops receiving after getting a valid packet. You will need to start it again with the following command:

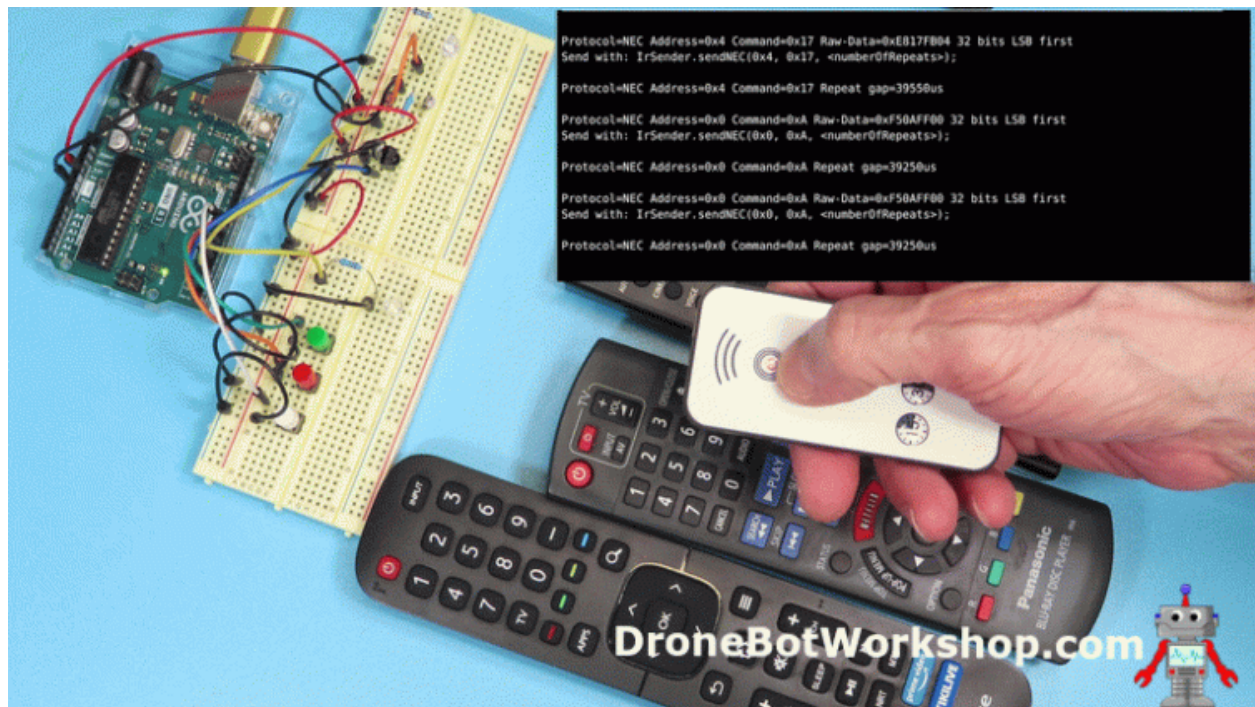
```
1 IrReceiver.resume();
```

The example also includes a framework for performing an action when a specific command value is received. You can add code here later if you wish.

Running SimpleReceiver

Load the sketch to the Arduino and open the serial monitor. Now aim a remote control at the receiver module and press a button on it.

If it was an NEC protocol remote (the most common), then you will get information regarding the protocol, address, and protocol on the serial monitor.



If it was another protocol, however, you will just get some raw data and a message saying, *"Received noise or an unknown (or not yet enabled) protocol"*.

Assuming that you know what the protocol should be, you can un-remark its line at the beginning of the sketch. In the video accompanying this article, I show examples of enabling it to recognize Sony and Panasonic remotes.

SimpleSender Example Code

Now we can move on to the *SimpleSender* sketch, and I'll bet that you can guess what it does! It's essentially the sending counterpart of the previous sketch.

The sketch will use the NEC protocol and send out command, address, and repeat codes. It will step through these, so if you leave it in a room with some remote-controlled devices, you may eventually affect one of them!

```
1 /*
2 * SimpleSender.cpp
3 *
4 * Demonstrates sending IR codes in standard format with address and command
5 * An extended example for sending can be found as SendDemo.
6 *
7 * Copyright (C) 2020-2022 Armin Joachimsmeier
8 * armin.joachimsmeier@gmail.com
9 *
10 * This file is part of Arduino-IRremote
11 https://github.com/Arduino-IRremote/Arduino-IRremote.
12 *
13 * MIT License
14 */
15 #include <Arduino.h>
16
17 #define DISABLE_CODE_FOR_RECEIVER // Disables restarting receiver after each send.
18 // Saves 450 bytes program memory and 269 bytes RAM if receiving functions are not
19 // used.
20 // #define SEND_PWM_BY_TIMER // Disable carrier PWM generation in software and
21 // use (restricted) hardware PWM.
22 // #define USE_NO_SEND_PWM // Use no carrier PWM, just simulate an active low
23 // receiver signal. Overrides SEND_PWM_BY_TIMER definition
24
25 /*
26 * This include defines the actual pin number for pins like IR_RECEIVE_PIN,
27 * IR_SEND_PIN for many different boards and architectures
28 */
29 #include "PinDefinitionsAndMore.h"
30
31 #include <IRremote.hpp> // include the library
```



```
2 void setup() {
2     pinMode(LED_BUILTIN, OUTPUT);
2
3
2     Serial.begin(115200);
4
2     // Just to know which program is running on my Arduino
5
2     Serial.println(F("START " __FILE__ " from " __DATE__ "\r\nUsing library version
2 " VERSION_IRREMOTE));
6
2     Serial.print(F("Send IR signals at pin "));
7     Serial.println(IR_SEND_PIN);
2
8
2     /*
9         * The IR library setup. That's all!
3         */
0 //     IrSender.begin(); // Start with IR_SEND_PIN as send pin and if
3 NO_LED_FEEDBACK_CODE is NOT defined, enable feedback LED at default feedback LED pin
1     IrSender.begin(DISABLE_LED_FEEDBACK); // Start with IR_SEND_PIN as send pin and
3 disable feedback LED at default feedback LED pin
2 }
3
3
3 /*
3
4 * Set up the data to be sent.
3
3 * For most protocols, the data is build up with a constant 8 (or 16 byte) address
5
3 * and a variable 8 bit command.
6
3 * There are exceptions like Sony and Denon, which have 5 bit address.
3 */
7 uint8_t sCommand = 0x34;
3
3 uint8_t sRepeats = 0;
8
3
9 void loop() {
```

```
4      /*
0          * Print current send values
4      */
1
4      Serial.println();
2      Serial.print(F("Send now: address=0x00, command=0x"));
4      Serial.print(sCommand, HEX);
3      Serial.print(F(", repeats="));
4      Serial.print(sRepeats);
4      Serial.println();
5
4      Serial.println(F("Send standard NEC with 8 bit address"));
6      Serial.flush();
4
7
4      // Receiver output for the first loop must be: Protocol=NEC Address=0x102
8      Command=0x34 Raw-Data=0xCB340102 (32 bits)
4      IrSender.sendNEC(0x00, sCommand, sRepeats);
9
5
0      /*
5          * Increment send values
1      */
5
2      sCommand += 0x11;
5      sRepeats++;
3      // clip repeats at 4
5      if (sRepeats > 4) {
4          sRepeats = 4;
5      }
5
5
6      delay(1000); // delay must be greater than 5 ms (RECORD_GAP_MICROS), otherwise
5 the receiver sees it as one long signal
0
```

For more projects and tutorials visit the DroneBot Workshop - <https://dronebotworkshop.com>

```
}
```

This sketch contains some constants that must be defined before the library is called. One of them, `DISABLE_CODE_FOR_RECEIVER`, is enabled. It disables the receiver function, restarting after each send, saving about 450 bytes of memory.

In Setup, we define the built-in LED as an output, which may seem odd but is due to its possible use later on in the code. It can be set to flash whenever an IR signal is sent, providing a visual indication of operation.

We start the IR sender with the following line:

```
1 IrSender.begin(DISABLE_LED_FEEDBACK);
```

Note that LED feedback is disabled; you can enable it if you wish to see the effect on the built-in LED.

We also define a few variables after Setup, a bit of an odd location, but perfectly valid. These are the “seed” values for the incrementing IR codes and repeat values.

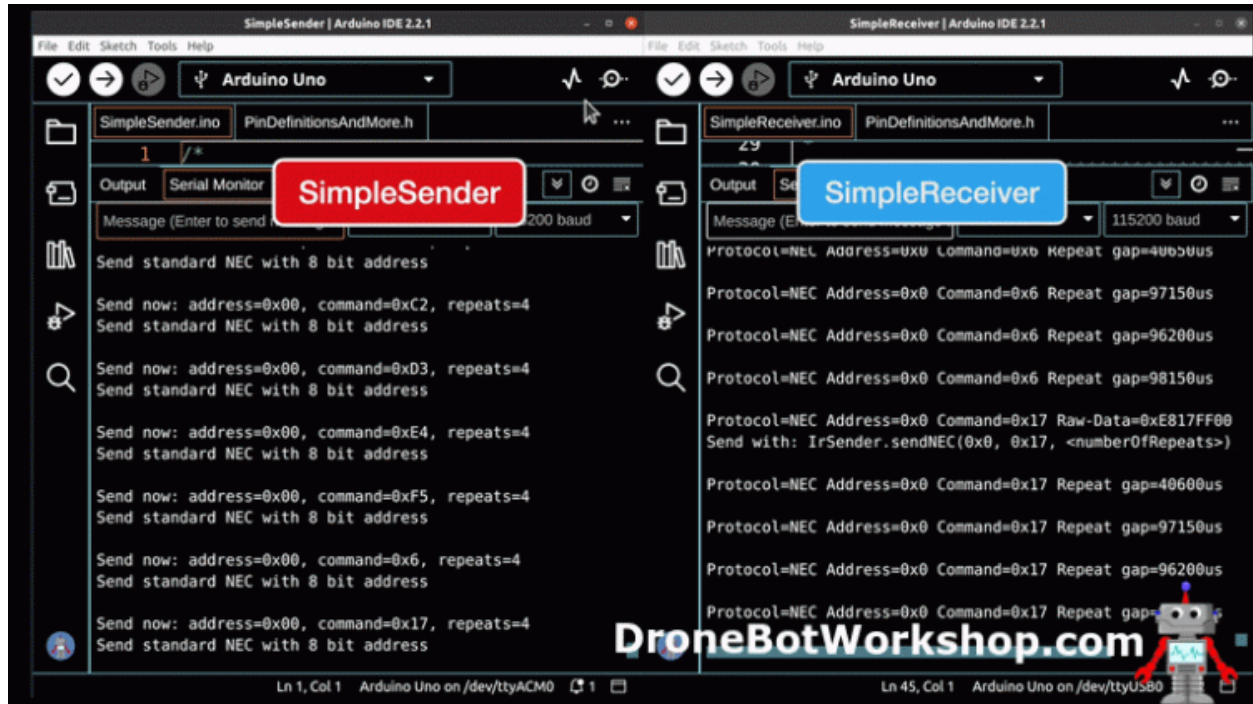
In the Loop, we print out the current values, then send them within this command:

```
1 IrSender.sendNEC(0x00, sCommand, sRepeats);
```

The first parameter is the Address, which is always 00 in this example. Feel free to change it.

After sending, we alter the values, delay a second, and do it again.

For more projects and tutorials visit the DroneBot Workshop - <https://dronebotworkshop.com>



Load the code up to your Arduino and observe the serial monitor. You can also run a second Arduino, as I did, with *SimpleReceiver*. That will allow you to monitor the results.

<https://dronebotworkshop.com>

IRMP Library

The *IRMP*, or [Infrared Multi Protocol Decoder + Encoder Library](#), is an alternative to the *Arduino-IRremote* library.

This library has a lower memory footprint than the other library, and it supports 50 different protocols. The *Arduino-IRremote* library only supports 17 protocols as of this writing.

You can install the IRMP Library using the Library Manager in the Arduino IDE. Once it is installed, it will also install several examples; most of these are reworked versions of the *Arduino-IRremote* library examples. This is great as you can use them to see the differences between the two libraries.

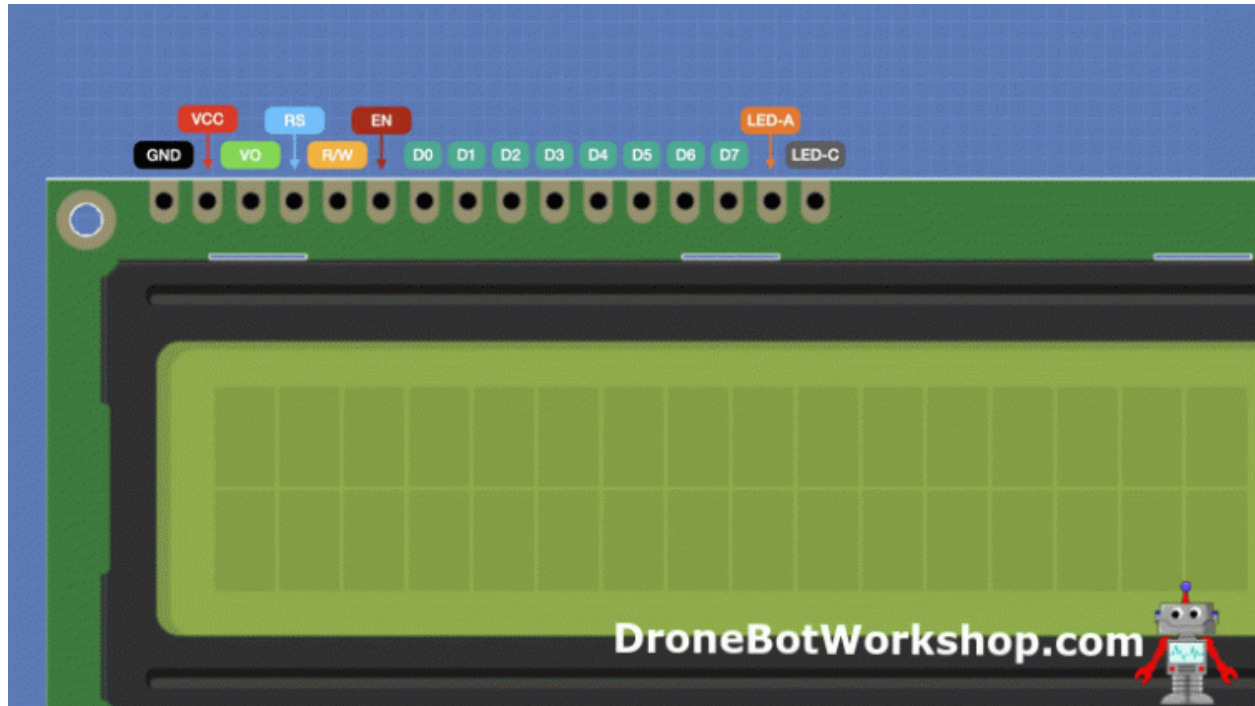
The wiring we hooked up on our Arduino will also work with the IRMP library examples.

AllProtocols LCD Hookup

We are going to run an example that is actually in both libraries, the AllProtocols example. This example reads IR protocols, along with address and command values, and displays them on a two-line LCD.

The example uses a 1602 LCD module, along with an Arduino. You'll also need a 10k pot to use as a contrast control for the LCD.

Here is the pinout of a standard 1602 display module. Note that your module may have slightly different pin designations for power and ground.



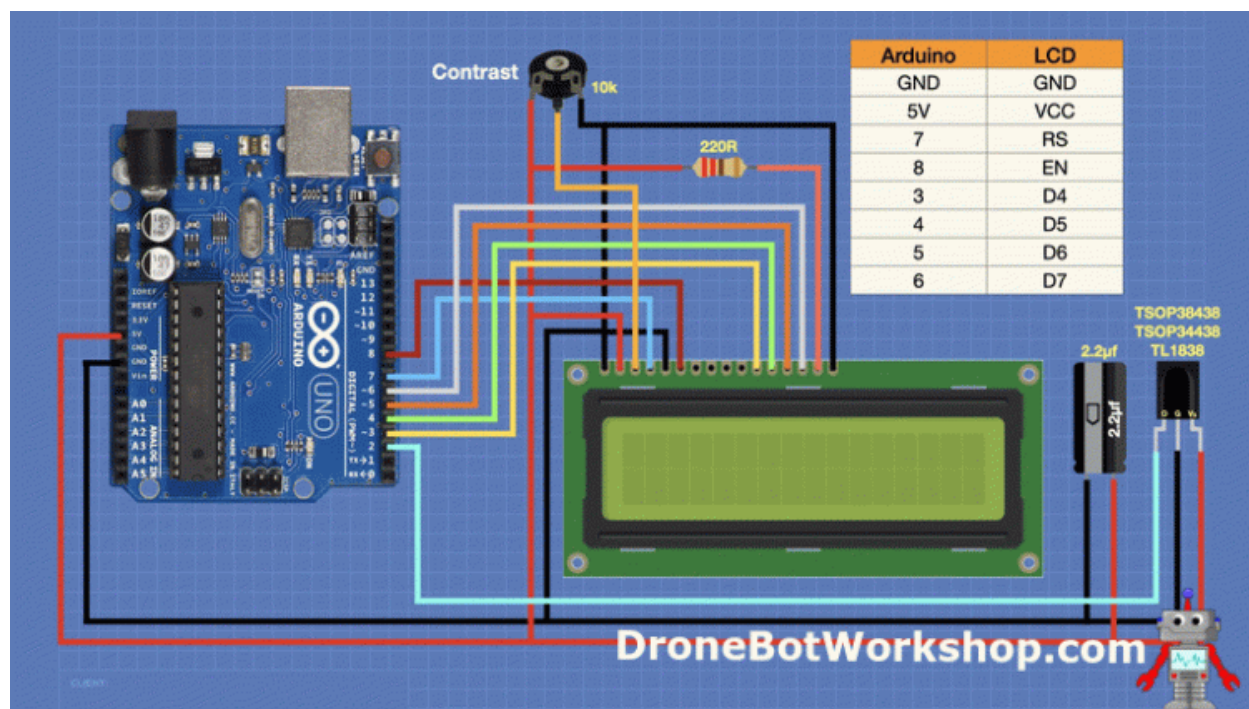
There are three ways you can interface a microcontroller to a 1602 LCD module:

- Parallel data, 8-bits
- Parallel data, 4-bits
- Serial data, I2C adapter (requires additional hardware)

Although the I2C adapter uses the least wiring, we will be hooking up our display in parallel. We will be using 4-bit mode to reduce the number of connections. This method is preferred in the AllProtocols example; a serial hookup would be missing some data (according to the notes in the sketch).

Here is how we will be hooking up our 1602 display. You could also use a 1604 display; the pinout is identical.

For more projects and tutorials visit the DroneBot Workshop - <https://dronebotworkshop.com>



AllProtocols Code & Demo

The *AllProtocols* sketch is one of the example sketches that was installed when you added the IRMP library to your Arduino IDE.

<https://dronebotworkshop.com>

```
1 /*
2  *   AllProtocols.cpp
3  *
4  *   Accepts 40 protocols concurrently
5  *   If you specify F_INTERRUPTS to 20000 at line 86 (default is 15000) it supports
6  *   LEGO + RCMM protocols, but disables PENTAX and GREE protocols.
7  *   if you see performance issues, you can disable MERLIN Protocol at line 88.
8  *
9  *   Uses a callback function which is called every time a complete IR command was
10  *   received.
11  *
12  *   Prints data to LCD connected parallel at pin 4-9 or serial at pin A4, A5
13  *
14  *   Copyright (C) 2019-2022 Armin Joachimsmeier
15  *   armin.joachimsmeier@gmail.com
16  *
17  *   This file is part of IRMP https://github.com/IRMP-org/IRMP.
18  *
19  *   IRMP is free software: you can redistribute it and/or modify
20  *   it under the terms of the GNU General Public License as published by
21  *   the Free Software Foundation, either version 3 of the License, or
22  *   (at your option) any later version.
23  *
24  *   This program is distributed in the hope that it will be useful,
25  *   but WITHOUT ANY WARRANTY; without even the implied warranty of
26  *   MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.
27  *   See the GNU General Public License for more details.
28  *
29  *   You should have received a copy of the GNU General Public License
30  *   along with this program. If not, see <http://www.gnu.org/licenses/gpl.html>.
```

```
2 *
2
2 */
2
3
2 #include <Arduino.h>
4
2 #include "PinDefinitionsAndMore.h"
5
2
6 /*
2 * Set input pin and output pin definitions etc.
7
2 */
8 #define IRMP_PROTOCOL_NAMES          1 // Enable protocol number mapping to
protocol strings - needs some program memory ~ 420 bytes here
2
9 #define IRMP_USE_COMPLETE_CALLBACK    1 // Enable callback functionality
10
3 // #define NO_LED_FEEDBACK_CODE      // Activate this if you want to suppress LED
0 feedback or if you do not have a LED. This saves 14 bytes code and 2 clock cycles
per interrupt.
3
1
3 #if __SIZEOF_INT__ == 4
2 #define F_INTERRUPTS                  20000 // Instead of default 15000 to support
3 LEGO + RCMM protocols
3
3 #else
3 // #define F_INTERRUPTS                20000 // Instead of default 15000 to support
4 LEGO + RCMM protocols, but this in turn disables PENTAX and GREE protocols :-(
3
3 // #define IRMP_32_BIT                  1 // This enables MERLIN protocol, but
5 decreases performance for AVR.
3
3 #endif
6
3
7 #include <irmpSelectAllProtocols.h> // This enables all possible protocols
3
3 // #define IRMP_SUPPORT_SIEMENS_PROTOCOL 1
8
3
9 /*
```



```
4 * After setting the definitions we can include the code and compile it.
0
4 */
1 #include <irmp.hpp>
4
2 IRMP_DATA irmp_data;
4
3
4 /*
4 * Activate the type of LCD you use
4 * Default is parallel LCD with 2 rows of 16 characters (1602).
5
4 * Serial LCD has the disadvantage, that the first repeat is not detected,
4
6 * because of the long lasting serial communication.
4
4 */
7 // #define USE_NO_LCD
4
4 // #define USE_SERIAL_LCD
8
4 /*
9 * Define the size of your LCD
5
4 */
0
4 // #define USE_2004_LCD
5
1 #if defined(USE_2004_LCD)
5
5 // definitions for a 2004 LCD
2
2 #define LCD_COLUMNS 20
5
5 #define LCD_ROWS 4
3
5 #else
5
4 #define USE_1602_LCD
5
5 // definitions for a 1602 LCD
5
5 #define LCD_COLUMNS 16
5
6 #define LCD_ROWS 2
5
5 #endif
7
```

```
5
8
9 #if defined(USE_SERIAL_LCD)
5
9 #include <LiquidCrystal_I2C.h> // Use an up to date library version, which has the
  init method
6
0 LiquidCrystal_I2C myLCD(0x27, LCD_COLUMNS, LCD_ROWS); // set the LCD address to
  0x27 for a 16 chars and 2 line display
6
1
6 #elif !defined(USE_NO_LCD)
2
2 #include <LiquidCrystal.h>
6
3 #define USE_PARALLEL_LCD
3
  //LiquidCrystal myLCD(4, 5, 6, 7, 8, 9);
6
4 LiquidCrystal myLCD(7, 8, 3, 4, 5, 6);
6 #endif
5
6
6 #if defined(USE_SERIAL_LCD) || defined(USE_PARALLEL_LCD)
6
6 #define USE_LCD
7
  # if defined(ADC_UTILS_ARE_AVAILABLE)
6
  // For cyclically display of VCC
8
  #include "ADCUtils.hpp"
6
9 #define MILLIS_BETWEEN_VOLTAGE_PRINT 5000
7
0 uint32_t volatile sMillisOfLastVoltagePrint;
0
  # endif
7
1
7 void printIRResultOnLCD();
2
2 size_t printHex(uint16_t aHexByteValue);
7
3 #endif
3
7
4 void handleReceivedIRData();
7
5
```

```
7 bool volatile sIRMPDataAvailable = false;
6
7
7 void setup()
7 {
8     Serial.begin(115200);
7
7 #if defined(__AVR_ATmega32U4__) || defined(SERIAL_PORT_USBVIRTUAL) ||
9 defined(SERIAL_USB) /*stm32duino*/ || defined(USBCON) /*STM32_stm32*/ ||
8 defined(SERIALUSB_PID) || defined(ARDUINO_attiny3217) \
0 || defined(__AVR_ATtiny1616__) || defined(__AVR_ATtiny3216__) ||
8 defined(__AVR_ATtiny3217__)
1     delay(4000); // To be able to connect Serial monitor after reset or power on and
8 before first printout
2 #endif
8
3     // Just to know which program is running on my Arduino
8     Serial.println(F("START " __FILE__ " from " __DATE__ "\r\nUsing library version
4 " VERSION_IRMP));
8
5     irmp_init();
8
8     irmp_irsnd_LEDFeedback(true); // Enable receive signal feedback at LED_BUILTIN
6
6     irmp_register_complete_callback_function(&handleReceivedIRData);
8
7
8     Serial.print(F("Ready to receive IR signals of protocols: "));
8
8     irmp_print_active_protocols(&Serial);
8
8     Serial.println(F("at pin " STR(IRMP_INPUT_PIN)));
9
9 #if defined(USE_SERIAL_LCD)
0     Serial.println(F("With serial LCD connection, the first repeat is not detected,
9 because of the long lasting serial communication!"));
9
1 #endif
9
2
2 #if defined(USE_LCD) && defined(ADC_UTILS_ARE_AVAILABLE)
9
3     getVCCVoltageMillivoltSimple(); // to initialize ADC mux and reference
```

```
9 #endif
4
9
5 #if defined(USE_SERIAL_LCD)
9     myLCD.init();
6     myLCD.clear();
9     myLCD.backlight();
7
#endif
9
8 #if defined(USE_PARALLEL_LCD)
9     myLCD.begin(LCD_COLUMNS, LCD_ROWS); // This also clears display
9
#endif
1
0
0 #if defined(USE_LCD)
1     myLCD.setCursor(0, 0);
0
1     myLCD.print(F("IRMP all v" VERSION_IRMP));
1
1     myLCD.setCursor(0, 1);
0
2     myLCD.print(F(__DATE__));
1
#endif
0
3
1
0 void loop()
4
{
1
0     if (sIRMPDataAvailable)
5     {
1
0         sIRMPDataAvailable = false;
6
1
0
7     /*
0
7     * Serial output
0
    * takes 2 milliseconds at 115200

```

```
1      */
0
8      irmp_result_print(&irmp_data);
1
0
9      #if defined(USE_LCD)
1
1      #   if defined(USE_SERIAL_LCD)
1
1          // This suppresses the receive of the 1. NEC repeat
0
1          disableIRTimerInterrupt(); // disable timer interrupt, since it disturbs the
1 LCD serial output
1
1      #   endif
1
1          printIRResultOnLCD();
1
1      #   if defined(USE_SERIAL_LCD)
2
1          enableIRTimerInterrupt();
1
1      #   endif
3
1      #endif
1
1      }
4
1
1      #if defined(USE_LCD) && defined(ADC_UTILS_ARE_AVAILABLE)
1
5          /*
1
1          * Periodically print VCC
1
6          */
1
1          if (millis() - sMillisOfLastVoltagePrint > MILLIS_BETWEEN_VOLTAGE_PRINT)
1
1          {
7
1
1          sMillisOfLastVoltagePrint = millis();
1
1          uint16_t tVCC = getVCCVoltageMillivoltSimple();
8
1
1
1          char tVoltageString[5];
1
9          dtostrf(tVCC / 1000.0, 4, 2, tVoltageString);
1
1          myLCD.setCursor(11, 0);
2
0          myLCD.print(tVoltageString);
```

For more projects and tutorials visit the DroneBot Workshop - <https://dronebotworkshop.com>

```

1   myLCD.print('V');
2
1   }
2
1 #endif
2
2 }
2
1
2 /*
3
4  * Here we know, that data is available.
1
2  * Since this function is executed in Interrupt handler context, make it short and do
4 not use delay() etc.
2
1  * In order to enable other interrupts you can call interrupts() (enable interrupt
2 again) after getting data.
5
5 */
1
2 #if defined(ESP8266) || defined(ESP32)
6 void IRAM_ATTR handleReceivedIRData()
1
2 #else
7 void handleReceivedIRData()
1
2 #endif
2
8 {
1
2
2 #if defined(USE_LCD) && defined(ADC_UTILS_ARE_AVAILABLE)
9
10 // reset voltage display timer
1
2 sMillisOfLastVoltagePrint = millis();
0
2 #endif
1
3
1
2
1 /*
3
1  * Just print the data to Serial and LCD
3
2  */
1
2 irmp_get_data(&irmp_data);
1
3 sIRMPDataAvailable = true;
3

```

<https://dronebotworkshop.com>


```
1 }
3
4
1 #if defined(USE_LCD)
3
5 /*
6
8 * LCD output for 1602 and 2004 LCDs
9
11 * 40 - 55 Milliseconds per initial output for a 1602 LCD
12
14 * for a 2004 LCD the initial clearing adds 55 ms.
15
17 * The expander runs at 100 kHz :-(
18
20 * 8 milliseconds for 8 bit; 10 ms for 16 bit code output
21
23 * 3 milliseconds for repeat output
24
26 *
27
29 */
30
31 void printIRResultOnLCD()
32
33 {
34
36     static uint8_t sLastProtocolIndex;
37
38     static uint16_t sLastProtocolAddress;
39
40
41
42     # if (LCD_ROWS >= 4)
43
44         static uint8_t sLastCommandPrintPosition = 13;
45
46         const uint8_t tStartRow = 2;
47
48
49     # else
50
51         static uint16_t sLastCommand;
52
53         static uint8_t sLastCommandPrintPosition;
54
55
56         const uint8_t tStartRow = 0;
57
58         bool tDisplayWasCleared = false;
59
60     # endif
61
62 }
```

```
1
4
7  /*
1    * Print only if protocol or address has changed
4
8    */
1    if (sLastProtocolIndex != irmp_data.protocol || sLastProtocolAddress !=
1    irmp_data.address)
4
9    {
1
5        sLastProtocolIndex = irmp_data.protocol;
0
        sLastProtocolAddress = irmp_data.address;
1 #  if (LCD_ROWS >= 4)
5
1    // clear data lines
1
5    myLCD.setCursor(0, tStartRow);
1
5    myLCD.print(F("                "));
2
5    myLCD.setCursor(0, tStartRow + 1);
1
5    myLCD.print(F("                "));
3 #  else
1
5    myLCD.clear();
4    tDisplayWasCleared = true;
1 #  endif
5
5
1    /*
5    * Show protocol name
6
1    */
5    myLCD.setCursor(0, tStartRow);
7
1 #  if defined(__AVR__)
1
5    const char *tProtocolStringPtr = (char*)
8    pgm_read_word(&irmp_protocol_names[irmp_data.protocol]);
1
5    myLCD.print((__FlashStringHelper*) (tProtocolStringPtr));
5 #  else
9
```

```
1      myLCD.print(irmp_protocol_names[irmp_data.protocol]);
6
0 # endif

1
6      /*
1      * Show address
1
6      */
2      myLCD.setCursor(0, tStartRow + 1);
1      myLCD.print(F("A="));
6
3      printHex(irmp_data.address);
1
6
4 # if (LCD_COLUMNS > 16)
1      /*
6
5      * Print prefix of command here, since it is constant string
1
6      */
6      myLCD.setCursor(9, tStartRow + 1);
6
1      myLCD.print(F("C="));
6 # endif
7      }
1
6      else
8      {
1
6      /*
9      * Show or clear repetition flag
1
6      */
7
0 # if (LCD_COLUMNS > 16)
1      myLCD.setCursor(18, tStartRow + 1);
7
1 # else
1
1      myLCD.setCursor(15, tStartRow + 1);
7 # endif
2
```

```
1      if (irmp_data.flags & IRMP_FLAG_REPETITION)
7
3      {
1          myLCD.print('R');
7          return; // Since it is a repetition, printed data has not changed
4      }
1
7      else
5      {
1          myLCD.print(' ');
7
6      }
1
7      }
7
1      /*
7
8      * Command prefix
1
7      */
7      uint16_t tCommand = irmp_data.command;
9
1
8 # if (LCD_COLUMNS <= 16)
0
1      // check if prefix position must change
1
8      if (tDisplayWasCleared || (sLastCommand > 0x100 && tCommand < 0x100) ||
1 (sLastCommand < 0x100 && tCommand > 0x100))
1
1      {
8
2          sLastCommand = tCommand;
1
1      /*
8
3          * Print prefix for 8/16 bit commands
1
1      */
1
8          if (tCommand >= 0x100)
4
1          {
8
1              sLastCommandPrintPosition = 9;
5
1          }
```

```
1         else
8
6         {
1             myLCD.setCursor(9, tStartRow + 1);
8
7             myLCD.print(F("C="));
1
             sLastCommandPrintPosition = 11;
8
             }
8
        }
1
#   endif
9
1
/*
9
0     * Command data
1
    */
9
1     myLCD.setCursor(sLastCommandPrintPosition, tStartRow + 1);
1
    printHex(tCommand);
1
9
}
2
1
9 size_t printHex(uint16_t aHexByteValue) {
3
    myLCD.print(F("0x"));
1
9    size_t tPrintSize = 2;
4
    if (aHexByteValue < 0x10 || (aHexByteValue > 0x100 && aHexByteValue < 0x1000)) {
1
        myLCD.print('0'); // leading 0
9
5        tPrintSize++;
1
    }
9
6    return myLCD.print(aHexByteValue, HEX) + tPrintSize;
1
}
9
#endif // defined(USE_LCD)
7
1
9
8
```

As with the examples centered around the *Arduino-IRremote* library, this code (and several of the other IRMP examples) has a pin definition file as well.

A lot of the code is centered around the LCD module. Around line 60, you will see instructions for changing to a serial (I2C) LCD or not using an LCD and just using the serial monitor.

In line 70, you'll find a definition you can change to use a 4-line LCD module, the 2004 (20 x 4 display).

In Setup, look for the following lines:

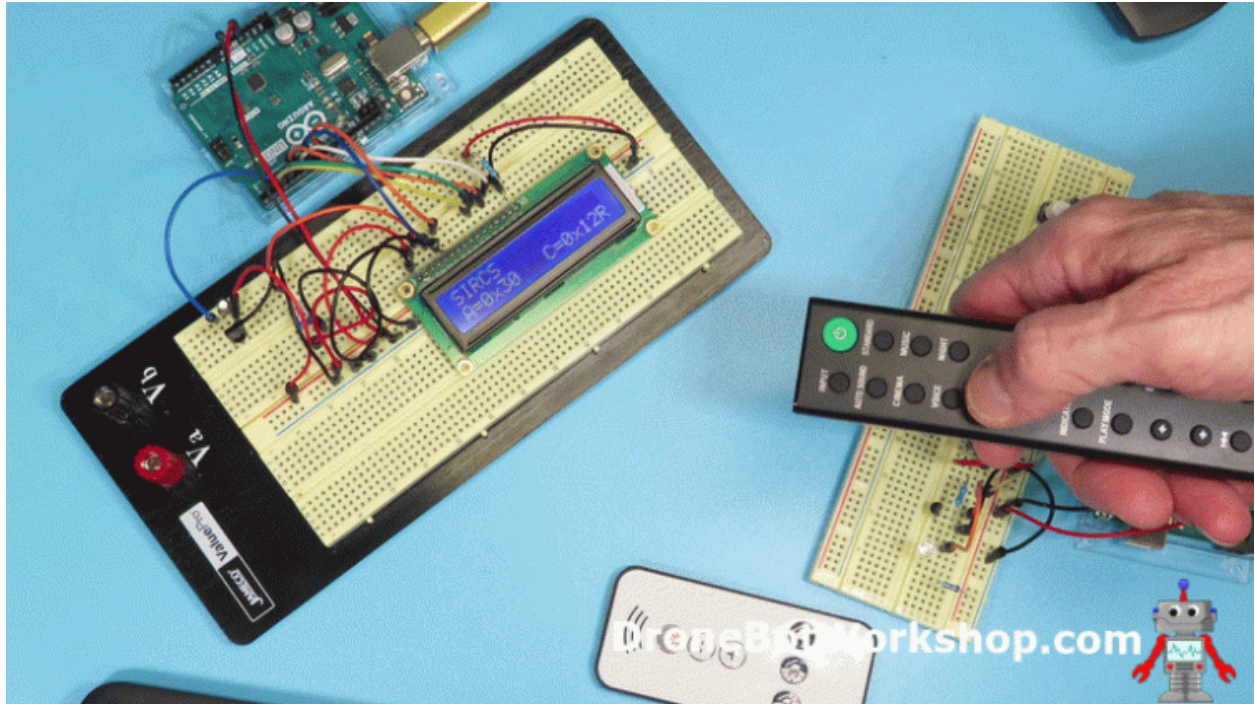
```
1 irmp_init();  
2 irmp_irsnd_LEDFeedback(true); // Enable receive signal feedback at LED_BUILTIN  
3 irmp_register_complete_callback_function(&handleReceivedIRData);
```

These lines illustrate how to use the IRMP library; it differs from the method used for the *Arduino-IRremote* library.

A function, *printIRResultOnLCD*, handles all the LCD formatting.

Load the code onto the Arduino, get out a remote, and try it out. You will likely have to adjust the contrast potentiometer for the best LCD visibility.

For more projects and tutorials visit the DroneBot Workshop - <https://dronebotworkshop.com>



This is a handy tool to have around when coding for remote controls.

<https://dronebotworkshop.com>

Emulating Transmitters and Receivers

By now, you should have a pretty good handle on using both libraries with IR receivers and emitters. So we can get to work and design or own remote-controlled devices.

To illustrate this, I have decided to “clone” (or perhaps “emulate” is a better word) an IR-controlled LED lamp. It’s a simple device with a simple IR remote control that uses NEC command codes. The remote has buttons for power, plus controls to increase or decrease the LED lamp brightness. It also has some timer buttons, but I will ignore them for this experiment.

We will use an Arduino to emulate the lamp first, controlling the brightness and status of the “user LED” we wired to pin D5.

After that, we will use the three push buttons to emulate the transmitter.

Between the two projects, you should have enough information and sample code to design your own custom remote-controller devices with an Arduino or other microcontroller. No additional wiring is needed, as we have everything we need on the existing Arduino setup.

Receiver Code

We’ll start with the receiver. We will receive the IR commands from the existing remote control and use them to control our “user LED.” It should operate in a similar fashion to our LED lamp.

We’ll use PWM (Pulse Width Modulation) to control the LED brightness. On the Arduino Uno, six of the I/O pins are PWM-capable, including pin D5, where our User LED is

For more projects and tutorials visit the DroneBot Workshop - <https://dronebotworkshop.com>

attached. The brightness will be set in increments, and you can adjust the value of these increments to suit your personal taste.

```
1  /*
2   IR Remote Emulator - Receiving
3   ir-remote-receive.ino
4   Control LED brightness with 3-button remote
5   Uses Arduino-IRremote Library -
6   https://github.com/Arduino-IRremote/Arduino-IRremote
7   Use SimpleReceiver example to determine button codes
8
9   DroneBot Workshop 2023
10  https://dronebotworkshop.com
11  */
12
13  // Specify protocol(s) - Must be listed before IRremote.hpp
14  #define DECODE_NEC
15
16  // Include required libraries
17  #include <Arduino.h>
18  #include <IRremote.hpp>
19
20  // IR Receiver pin
21  #define IR_RECEIVE_PIN 2
22
23  // LED Pin
24  #define LED_PIN 5
25
26  // Data received variable
27  volatile bool irDataReceived = false;
28
29  // Lamp variables
```

```
29 volatile bool lampPower = false;
30 volatile int lampLevel = 255;
31
32 // Button code variables (change to match your remote control)
33 uint16_t codePower = 0x0A;
34 uint16_t codeMinus = 0x09;
35 uint16_t codePlus = 0x0B;
36
37 // Receive Callback
38 void ReceiveCallbackHandler() {
39     // Decode IR data
40     IrReceiver.decode();
41
42     // See if it matches one of our control codes
43     if (IrReceiver.decodedIRData.command == codePower) {
44         // Power button
45         // Toggle lamp power variable
46         lampPower = !lampPower;
47
48     } else if (IrReceiver.decodedIRData.command == codeMinus) {
49         // Minus Button
50         // Only change level if lamp is on
51         if (lampPower) {
52             lampLevel = lampLevel - 16;
53             if (lampLevel < 8) {
54                 // Don't let level drop to zero
55                 lampLevel = 8;
56             }
57         }
58     }
59 }
```

```
57     }
58 } else if (IrReceiver.decodedIRData.command == codePlus) {
59     // Plus Button
60     // Only change level if lamp is on
61     if (lampPower) {
62         lampLevel = lampLevel + 16;
63         if (lampLevel > 255) {
64             lampLevel = 255;
65         }
66     }
67 }
68
69 // Set data received flag true
70 irDataReceived = true;
71
72 // Resume receiving
73 IrReceiver.resume();
74 }
75
76
77 void setup() {
78
79     // Serial monitor
80     Serial.begin(115200);
81
82     // Set LED as an output
83     pinMode(LED_PIN, OUTPUT);
84
```



```
85  // Set LED off
86  digitalWrite(LED_PIN, LOW);
87
88  // Start IR Receiver with LED feedback on BUILTIN LED
89  IrReceiver.begin(IR_RECEIVE_PIN, ENABLE_LED_FEEDBACK);
90
91  // Attach Callback Handler
92  IrReceiver.registerReceiveCompleteCallback(ReceiveCallbackHandler);
93 }
94
95 void loop() {
96  // See if new data is available
97  if (irDataReceived) {
98      // Print variable values
99      Serial.print("Power: ");
100     Serial.println(lampPower);
101     Serial.print("Level: ");
102     Serial.println(lampLevel);
103
104     // Set LED Power and Level
105     if (lampPower) {
106         // Lamp is powered, set level
107         analogWrite(LED_PIN, lampLevel);
108     } else {
109         // Lamp is off
110         digitalWrite(LED_PIN, LOW);
111     }
112 }
```

```
113     // Set data received flag false
114     irDataReceived = false;
115 }
116 }
```

We begin by defining the protocol(s) we wish to use. As my lamp is an NEC protocol device, this is what I have defined. If you are building an IR control from scratch, the NEC protocol is a good choice. Just be sure to set the address to a unique value to avoid conflicts with other IR remotes in the vicinity.

If you need to define a different protocol, see the *SimpleReceiver* example for a list of them.

We then include the library and define the pins for both the IR receiver and the User LED. After that, we define a boolean for the lamp power state and an integer for the level. These are both set volatile, as they are used in a callback function.

The next three lines are the codes for our buttons. If you are emulating another remote control, you will want to change these to match your control.

Next, we come to the Receive Callback function, which is called every time valid IR data is received.

In the callback function, we grab the button code and check it against the three code values we have defined.

If it matches the power key, then we toggle the boolean that represents the lamp state.

If it matches an up or down key, we increment or decrease the value of the LED PWM by 8 (you can change that if you like). We ensure that it does not go above 255 or below 8. I picked eight instead of zero, as I didn't want the lamp to completely dim. This is the way the lamp we are emulating works.

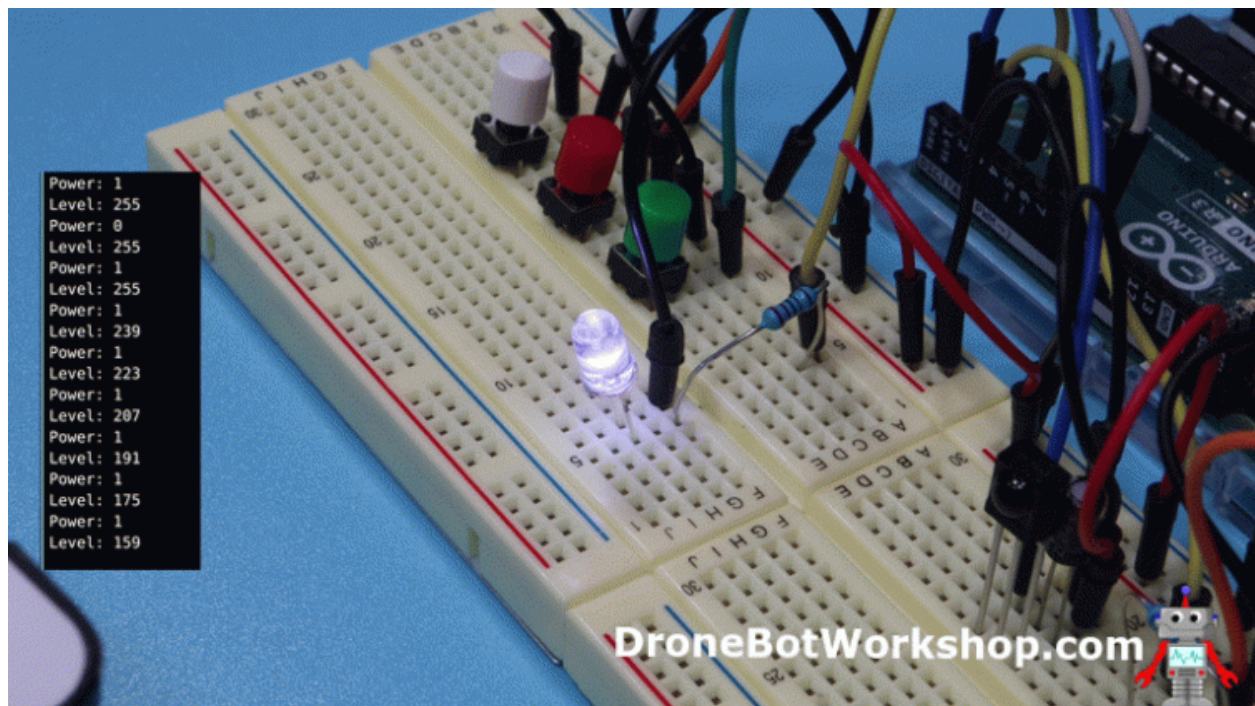
For more projects and tutorials visit the DroneBot Workshop - <https://dronebotworkshop.com>

In Setup, we start the serial monitor, define our LED as an output and turn it off, start the IR receiver, and attach the callback to the function we just described.

In the Loop, we see if there is new data. If it is, we examine the variables for both the LED status (on or off) and LED brightness. We use these to control the LED.

Note that we only change brightness when the LED status is “on”; if it is off, then the controls are recognized but ignored.

Load the sketch and get out your remote. If you coded the values correctly, you should be able to control the status and brightness of the LED.



Also note that if you turn the LED off, it retains its brightness level.

Transmitter Code

<https://dronebotworkshop.com>

For more projects and tutorials visit the DroneBot Workshop - <https://dronebotworkshop.com>

Now let's take a look at the code we will use for our transmitter, or "remote control".

We'll use our three push buttons to control the power and brightness of our LED lamp.

You can also use it to control the previous experiment if you have a second Arduino.

For more projects and tutorials visit the DroneBot Workshop - <https://dronebotworkshop.com>

```
1  /*
2   IR Remote Emulator - Sending
3   ir-remote-send.ino
4   Emulates three buttons of remote control
5   Uses TinySender Library packaged with Arduino-IRremote Library
6   -- https://github.com/Arduino-IRremote/Arduino-IRremote --
7   Uses pin-change interrupts for pushbuttons
8   Use SimpleReceiver example to determine button codes
9
10  DroneBot Workshop 2023
11  https://dronebotworkshop.com
12 */
13
14 // Include required libraries
15 #include <Arduino.h>
16 #include "TinyIRSender.hpp"
17
18 // Define IR LED pin
19 #define IR_SEND_PIN 3
20
21 // Define the pins for the buttons
22 const byte buttonPin1 = 10;
23 const byte buttonPin2 = 9;
24 const byte buttonPin3 = 8;
25
26 // Button code variables (change to match your remote control)
27 uint16_t codePower = 0x0A;
28 uint16_t codeMinus = 0x09;
```

<https://dronebotworkshop.com>

```
29 uint16_t codePlus = 0x0B;
30
31 // Variables to hold complete send parameters
32 uint16_t sAddress = 0x00;
33 volatile uint16_t sCommand = 0x0A;
34 uint16_t sRepeats = 0;
35
36 // Button pressed flag
37 volatile bool buttonPressed = false;
38
39 // Button pin-change service routine
40 ISR(PCINT0_vect) {
41     // Port B Interrupt occurred
42
43     // Only run if buttonPressed is false, as a debounce technique
44     if (!buttonPressed) {
45
46         // Check if this was D10 (button 1)
47         if (digitalRead(buttonPin1) == LOW) {
48             //Pin D10 triggered the ISR on a Falling pulse
49             sCommand = codePower;
50         }
51         // Check if this was D9 (button 2)
52         if (digitalRead(buttonPin2) == LOW) {
53             //Pin D9 triggered the ISR on a Falling pulse
54             sCommand = codeMinus;
55         }
56         // Check if this was D8 (button 3)
```

```
57     if (digitalRead(buttonPin3) == LOW) {
58         //Pin D8 triggered the ISR on a Falling pulse
59         sCommand = codePlus;
60     }
61
62     // Change button pressed flag
63     buttonPressed = true;
64 }
65 }
66
67 void setup() {
68
69     // Serial monitor
70     Serial.begin(115200);
71
72     // Set switches as inputs with pullups
73     pinMode(buttonPin1, INPUT_PULLUP);
74     pinMode(buttonPin2, INPUT_PULLUP);
75     pinMode(buttonPin3, INPUT_PULLUP);
76     // Built-in LED setup for IR library
77     pinMode(LED_BUILTIN, OUTPUT);
78
79     // Setup pin-change interrupt masks for port B (pins 8 - 13)
80     // Enable PCIE0 Bit0 = 1 (Port B)
81     PCICR |= B00000001;
82     // Enable PCINT0, PCINT1 & PCINT23 (Pins D8, D9 & D10)
83     PCMSK0 |= B00000111;
84 }
```



```
85
86 void loop() {
87   // See if a button has been pressed
88   if (buttonPressed) {
89     // Send IR code
90     sendNEC(IR_SEND_PIN, sAddress, sCommand, sRepeats);
91     // Print to serial monitor
92     Serial.print("Sent command ");
93     Serial.println(sCommand, HEX);
94     // Short delay to debounce
95     delay(100);
96     // Reset the flag
97     buttonPressed = false;
98   }
99   // Another short delay
100  delay(100);
101 }
```

There are a couple of interesting things about this sketch:

- It uses the *TinySender* library. This small library is included with the *Arduino-IRremote* library. It has a smaller footprint, but not as many features.
- It uses pin change interrupts to detect pushbutton presses.

We begin by including the Arduino and TinySender libraries.

Next, we define the pins for the IR emitter LED and for the three push buttons. After that are the button code variables; as with the receiver sketch, you can change these if you wish.

For more projects and tutorials visit the DroneBot Workshop - <https://dronebotworkshop.com>

The next three variables hold the IR data values that we will be sending. Pay note to the first one, the address. I am using hexadecimal 0x00, as this is what my remote lamp uses, but if you are emulating a different device, you will need to change this.

There is also a boolean that is set when a button has been pressed.

After defining the values, we move into the pin-change interrupt service routine. This will be called whenever one of the buttons in our pin-change group is called. There are three groups, and all of our push buttons are in group (or port) B , so any activity on any group B button triggers this interrupt.

In the interrupt service routine, we scan to see which button was pressed. We then set the value of the command variable to match the button, and we set the *buttonPressed* flag to true.

In Setup, we activate the Serial Monitor and define the buttons as inputs with internal pull-up resistors. We also define the built-in LED as an output.

After that, we set up pin-change interrupt masks. This enables the three pins we are using for our buttons. If you want to add buttons, you must change the mask. It's a simple binary pattern; look at it, and you will see the three I/O pins we are using.

In the Loop, we see if a button was pressed by checking the *buttonPressed* boolean. If it has, then we send the IR code using the current command value.

We add a few short delays to debounce and then look for a pressed button again.

Load the sketch and try it out! If you have coded the buttons with the correct values, you should be able to use them to control your IR device.

This can obviously be expanded upon to build bigger remotes.

Capturing IR Remote Codes

Up to this point, we have been decoding IR remote control signals and sending them with a known protocol. But what do you do if you don't know the protocol, or if you don't have a setting for it in your library?

Instead of decoding the IR signal, another option is to capture it and save the raw data. This data can then be used to resend the IR signal or to recognize it again.

There are a couple of examples included with the IRremote library that illustrate how to capture and work with raw IR data.

Both of the following examples perform the same function; however, they use different techniques to achieve it.

The functionality is as follows:

- The sketch sends and receives IR codes.
- By default, it is in receive mode.
- When an IR transmission is detected, it captures it and saves it. How it does this is different in each sketch.
- After saving the transmission, you can use a pushbutton to send it again. You can keep using the button to send it.
- If you are not sending data, then you are in receive mode.
- If a new IR packet is captured, then it is now the one sent when the button is pressed.

ReceiveAndSend Code

For more projects and tutorials visit the DroneBot Workshop - <https://dronebotworkshop.com>

In this sketch, we attempt to decode the incoming packet first, using known libraries. If we succeed, then we just store the protocol type, address, and command code for the button.

If we can't determine what protocol the incoming data is, we store the raw data for retransmission.

```
1 /*
2  * ReceiveAndSend.cpp
3  *
4  * Record and play back last received IR signal at button press.
5  * The logic is:
6  * If the button is pressed, send the IR code.
7  * If an IR code is received, record it.
8  * If the protocol is unknown or not enabled, store it as raw data for later sending.
9  *
10 * An example for simultaneous receiving and sending is in the UnitTest example.
11 *
12 * An IR detector/demodulator must be connected to the input IR_RECEIVE_PIN.
13 *
14 * A button must be connected between the input SEND_BUTTON_PIN and ground.
15 * A visible LED can be connected to STATUS_PIN to provide status.
16 *
17 *
18 * Initially coded 2009 Ken Shirriff http://www.righto.com
19 *
20 * This file is part of Arduino-IRremote
21 * https://github.com/Arduino-IRremote/Arduino-IRremote.
22 *
23 *
24 *****
25 * MIT License
26 *
27 * Copyright (c) 2009-2023 Ken Shirriff, Armin Joachimsmeier
28 *
29 * Permission is hereby granted, free of charge, to any person obtaining a copy
30 * of this software and associated documentation files (the "Software"), to deal
```

```
2 * in the Software without restriction, including without limitation the rights
2
2 * to use, copy, modify, merge, publish, distribute, sublicense, and/or sell
2
3 * copies of the Software, and to permit persons to whom the Software is furnished
2
2 * to do so, subject to the following conditions:
4
4 *
2
2 * The above copyright notice and this permission notice shall be included in all
5
2 * copies or substantial portions of the Software.
6
6 *
2
2 * THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
7 IMPLIED,
2
2 * INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A
8
2 * PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT
2
9 * HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION
OF
3
0 * CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE
SOFTWARE
3
1 * OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.
3
3 *
2
2 *****
3
3 */
3
#include <Arduino.h>
3
4
3
3 #include "PinDefinitionsAndMore.h" // Define macros for input and output pin etc.
5
3
6 /*
3
3 * Specify which protocol(s) should be used for decoding.
7
2 * If no protocol is defined, all protocols (except Bang&Olufsen) are active.
3
3 * This must be done before the #include <IRremote.hpp>
8
3
3 */
9
9 // #define DECODE_DENON          // Includes Sharp
```

```
4 // #define DECODE_JVC
0
// #define DECODE_KASEIKYO
4
1 // #define DECODE_PANASONIC // alias for DECODE_KASEIKYO
4
// #define DECODE_LG
2 #define DECODE_NEC // Includes Apple and Onkyo
4
// #define DECODE_SAMSUNG
3
// #define DECODE_SONY
4
4 // #define DECODE_RC5
4
4 // #define DECODE_RC6
5
4
6 // #define DECODE_BOSEWAVE
4
// #define DECODE_LEGO_PF
7
// #define DECODE_MAGIQUEST
4
// #define DECODE_WHYNTER
8
// #define DECODE_FAST
4
9 //
5 #if !defined(RAW_BUFFER_LENGTH)
0
# if RAMEND <= 0x4FF || RAMSIZE < 0x4FF
5
1 #define RAW_BUFFER_LENGTH 120
5
# elif RAMEND <= 0xAFF || RAMSIZE < 0xAFF // 0xAFF for LEONARDO
2 #define RAW_BUFFER_LENGTH 400 // 600 is too much here, because we have additional
5 uint8_t rawCode[RAW_BUFFER_LENGTH];
3
# else
5
# define RAW_BUFFER_LENGTH 750
4
# endif
5
5 #endif
5
6
// #define EXCLUDE_UNIVERSAL_PROTOCOLS // Saves up to 1000 bytes program memory.
5
7
```



```
5 // #define EXCLUDE_EXOTIC_PROTOCOLS // saves around 650 bytes program memory if all
8 other protocols are active

5 // #define NO_LED_FEEDBACK_CODE // saves 92 bytes program memory
9
6 // #define RECORD_GAP_MICROS 12000 // Default is 5000. Activate it for some LG air
6 conditioner protocols
0
6 // #define SEND_PWM_BY_TIMER // Disable carrier PWM generation in software and
6 use (restricted) hardware PWM.
1
6 // #define USE_NO_SEND_PWM // Use no carrier PWM, just simulate an active low
6 receiver signal. Overrides SEND_PWM_BY_TIMER definition
2
6
3 // MARK_EXCESS_MICROS is subtracted from all marks and added to all spaces before
6 decoding,
4
6 // to compensate for the signal forming of different IR receiver modules. See also
4 IRremote.hpp line 142.
6
5 #define MARK_EXCESS_MICROS 20 // Adapt it to your IR receiver module. 20 is
6 recommended for the cheap VS1838 modules.
6
6
6 // #define DEBUG // Activate this for lots of lovely debug output from the decoders.
7
6
8 #include <IRremote.hpp>
6
9 int SEND_BUTTON_PIN = APPLICATION_PIN;
7
0
int DELAY_BETWEEN_REPEAT = 50;
7
1
7 // Storage for the recorded code
2
struct storedIRDataStruct {
7
    IRData receivedIRData;
3
    // extensions for sendRaw
7
    uint8_t rawCode[RAW_BUFFER_LENGTH]; // The durations if raw
4
    uint8_t rawCodeLength; // The length of the code
7
5
```

```
7 } sStoredIRData;
6
7
7 bool sSendButtonWasActive;
7
7
8 void storeCode();
7
7 void sendCode(storedIRDataStruct *aIRDataToSend);
9
8
0 void setup() {
8     pinMode(SEND_BUTTON_PIN, INPUT_PULLUP);
1
8
2     Serial.begin(115200);
8
3 #if defined(__AVR_ATmega32U4__) || defined(SERIAL_PORT_USBVIRTUAL) ||
3 defined(SERIAL_USB) /*stm32duino*/ || defined(USBCON) /*STM32_stm32*/ ||
    defined(SERIALUSB_PID) || defined(ARDUINO_attiny3217)
8
4     delay(4000); // To be able to connect Serial monitor after reset or power up and
    before first print out. Do not wait for an attached Serial Monitor!
8
5 #endif
8
    // Just to know which program is running on my Arduino
6
    Serial.println(F("START " __FILE__ " from " __DATE__ "\r\nUsing library version
8 " VERSION_IRREMOTE));
7
8
8 // Start the receiver and if not 3. parameter specified, take LED_BUILTIN pin
    from the internal boards definition as default feedback LED
8
    IrReceiver.begin(IR_RECEIVE_PIN, ENABLE_LED_FEEDBACK);
9
    Serial.print(F("Ready to receive IR signals of protocols: "));
9
    printActiveIRProtocols(&Serial);
0
    Serial.println(F("at pin " STR(IR_RECEIVE_PIN)));
1
9
2 IrSender.begin(); // Start with IR_SEND_PIN as send pin and enable feedback LED
    at default feedback LED pin
9
3
```

```
9   Serial.print(F("Ready to send IR signals at pin " STR(IR_SEND_PIN) " on press of
4 button at pin "));

9   Serial.println(SEND_BUTTON_PIN);
5 }

9
6
9 void loop() {
7
9   // If button pressed, send the code.
8
9   bool tSendButtonIsActive = (digitalRead(SEND_BUTTON_PIN) == LOW); // Button pin
9 is active LOW
9
1
0  /*
0   * Check for current button state
1
0   */
1   if (tSendButtonIsActive) {
1
0     if (!sSendButtonWasActive) {
2       Serial.println(F("Stop receiving"));
1
0       IrReceiver.stop();
3     }
1
0     /*
0     * Button pressed -> send stored data
4
0     */
1     Serial.print(F("Button pressed, now sending "));
5     if (sSendButtonWasActive == tSendButtonIsActive) {
1       Serial.print(F("repeat "));
0
6       sStoredIRData.receivedIRData.flags = IRDATA_FLAGS_IS_REPEAT;
1     } else {
0
7       sStoredIRData.receivedIRData.flags = IRDATA_FLAGS_EMPTY;
    }
  }
```

For more projects and tutorials visit the DroneBot Workshop - <https://dronebotworkshop.com>

```
1      Serial.flush(); // To avoid disturbing the software PWM generation by serial
0 output interrupts
8
1      sendCode(&sStoredIRData);
1
0      delay(DELAY_BETWEEN_REPEAT); // Wait a bit between retransmissions
9
1
1      } else if (sSendButtonWasActive) {
0
1      /*
1      * Button is just released -> activate receiving
1
1      */
1
1      // Restart receiver
1
1      Serial.println(F("Button released -> start receiving"));
2
1      IrReceiver.start();
1
1
3
1      } else if (IrReceiver.decode()) {
1
1      /*
4
1      * Button is not pressed and data available -> store received data and resume
1
1      */
5
1      storeCode();
1
1      IrReceiver.resume(); // resume receiver
6
1      }
1
1
7      sSendButtonWasActive = tSendButtonIsActive;
1
1      delay(100);
1
1 }
8
1
1 // Stores the code for later playback in sStoredIRData
9
1 // Most of this code is just logging
1
1 void storeCode() {
2
0      if (IrReceiver.decodedIRData.rawDataPtr->rawlen < 4) {
```

<https://dronebotworkshop.com>

```
1      Serial.print(F("Ignore data with rawlen="));
2
1      Serial.println(IrReceiver.decodedIRData.rawDataPtr->rawlen);
2
1      return;
2
2    }
2
1    if (IrReceiver.decodedIRData.flags & IRDATA_FLAGS_IS_REPEAT) {
2
1      Serial.println(F("Ignore repeat"));
2
3      return;
4
1    }
2
4    if (IrReceiver.decodedIRData.flags & IRDATA_FLAGS_IS_AUTO_REPEAT) {
1
2      Serial.println(F("Ignore autorepeat"));
2
5      return;
6
1    }
2
6    if (IrReceiver.decodedIRData.flags & IRDATA_FLAGS_PARITY_FAILED) {
1
2      Serial.println(F("Ignore parity error"));
2
7      return;
8
1    }
2
8    /*
1      * Copy decoded data
2
2    */
9
1    sStoredIRData.receivedIRData = IrReceiver.decodedIRData;
2
3
0    if (sStoredIRData.receivedIRData.protocol == UNKNOWN) {
1
2      Serial.print(F("Received unknown code and store "));
3
1      Serial.print(IrReceiver.decodedIRData.rawDataPtr->rawlen - 1);
2
1      Serial.println(F(" timing entries as raw "));
3
2      IrReceiver.printIRResultRawFormatted(&Serial, true); // Output the results
2 in RAW format
1
1      sStoredIRData.rawCodeLength = IrReceiver.decodedIRData.rawDataPtr->rawlen -
3 1;
3
```

```
1      /*
3
4      * Store the current raw data in a dedicated array for later usage
5
6      */
7
8      IrReceiver.compensateAndStoreIRResultInArray(sStoredIRData.rawCode);
9
10     } else {
11
12         IrReceiver.printIRResultShort(&Serial);
13
14         IrReceiver.printIRSendUsage(&Serial);
15
16         sStoredIRData.receivedIRData.flags = 0; // clear flags -esp. repeat- for
17 later sending
18
19         Serial.println();
20
21     }
22 }
23
24
25 void sendCode(storedIRDataStruct *aIRDataToSend) {
26
27     if (aIRDataToSend->receivedIRData.protocol == UNKNOWN /* i.e. raw */) {
28
29         // Assume 38 KHz
30
31         IrSender.sendRaw(aIRDataToSend->rawCode, aIRDataToSend->rawCodeLength, 38);
32
33
34         Serial.print(F("raw "));
35
36         Serial.print(aIRDataToSend->rawCodeLength);
37
38         Serial.println(F(" marks or spaces"));
39
40     } else {
41
42
43         /*
44
45         * Use the write function, which does the switch for different protocols
46
47         */
48
49         IrSender.write(&aIRDataToSend->receivedIRData);
50
51         printIRResultShort(&Serial, &aIRDataToSend->receivedIRData, false);
52
53     }
54 }
```

For more projects and tutorials visit the DroneBot Workshop - <https://dronebotworkshop.com>

```
1 }  
4
```

The sketch starts like many of the other examples, with a list of protocols where only the NEC protocol is not remarked out.

We then define a buffer whose size is dependent upon the memory available. This is determined using constants in the pin definitions file.

After including the library, we define the pin for our send push button. **This is where we need to change the code to match our wiring.**

Change the line

```
1 int SEND_BUTTON_PIN = APPLICATION_PIN;
```

to read

```
1 int SEND_BUTTON_PIN = 10;
```

This will have us use PB1 as our push button.

We then build a data structure to store the recorded code. It has elements for the decoded IR data, as well as raw IR data, which we use for devices that we can't decode. We also define a boolean to indicate that the send button was active.

In Setup, we set the button pin to an input with an internal pull-up resistor, and we set up the serial monitor. We start both the IR receiver and IR sender and print the status to the serial monitor.

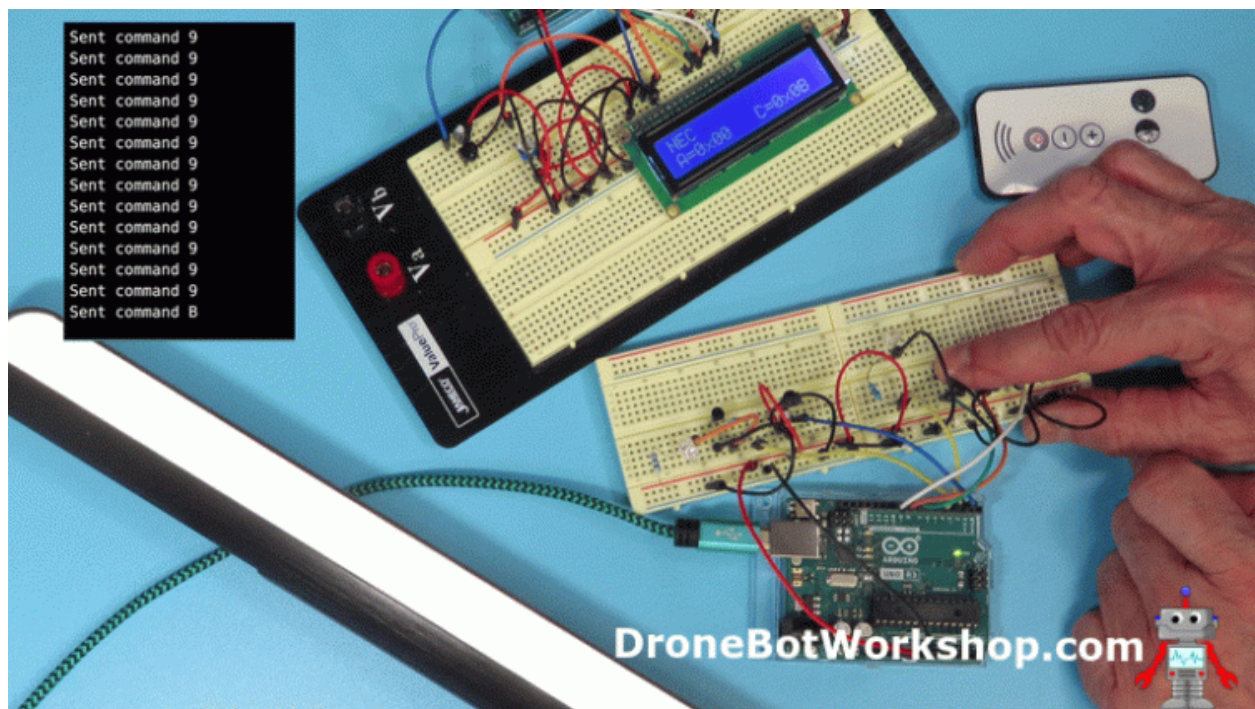
In the Loop, we check to see if the button was pressed. If it was, then we stop receiving and send the stored code.

For more projects and tutorials visit the DroneBot Workshop - <https://dronebotworkshop.com>

There is also a function named *storeCode*, which handles most of the receiver tasks. The function checks to see if it can decode the incoming data. If it can't decide the protocol, then it stores the timing entries as raw data.

Load the code to the Arduino. If you happen to still have the AllProtocols LCD demo assembled, it can be handy to test this with. Otherwise, you can test with a remote and its target device.

Aim the remote control at the demo and press a button. You should observe the captured data on the serial monitor. If it is a known protocol, you will see the decoded data; if it is unknown, then you'll see some raw data and statistics.



Now, the data has been captured. Press button PB1. This should send the captured data out via the IR emitter LED.

This technique could be the basis of a number of advanced experiments.

<https://dronebotworkshop.com>

ReceiveAndSendDistanceWidth Code

This sketch performs the same function as the previous one, but this time, we don't bother trying to decode the signal. We also use a different method of capturing raw data, one that can work with many more controls.

```
1 /*
2  * ReceiveAndSendDistanceWidth.cpp
3  *
4  * Record and play back last received distance width IR signal at button press.
5  * Using DistanceWidthProtocol covers a lot of known and unknown IR protocols,
6  * and requires less memory than raw protocol.
7  *
8  * The logic is:
9  * If the button is pressed, send the IR code.
10 * If an IR code is received, record it.
11 *
12 * An example for simultaneous receiving and sending is in the UnitTest example.
13 *
14 * An IR detector/demodulator must be connected to the input IR_RECEIVE_PIN.
15 *
16 * A button must be connected between the input SEND_BUTTON_PIN and ground.
17 * A visible LED can be connected to STATUS_PIN to provide status.
18 *
19 *
20 * This file is part of Arduino-IRremote
21 * https://github.com/Arduino-IRremote/Arduino-IRremote.
22 *
23 *****
24 * MIT License
25 *
26 * Copyright (c) 2023 Armin Joachimsmeier
27 *
28 * Permission is hereby granted, free of charge, to any person obtaining a copy
29 * of this software and associated documentation files (the "Software"), to deal
```

```
2 * in the Software without restriction, including without limitation the rights
2
2 * to use, copy, modify, merge, publish, distribute, sublicense, and/or sell
2
3 * copies of the Software, and to permit persons to whom the Software is furnished
2
2 * to do so, subject to the following conditions:
4
4 *
2
2 * The above copyright notice and this permission notice shall be included in all
5
2 * copies or substantial portions of the Software.
6
6 *
2
2 * THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
7 IMPLIED,
2
2 * INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A
8
2 * PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT
2
9 * HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION
OF
3
0 * CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE
SOFTWARE
3
1 * OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.
3
3 *
2
2 *****
3
3 */
3
3 #include <Arduino.h>
3
4
3 #include "PinDefinitionsAndMore.h" // Define macros for input and output pin etc.
5
5 #if !defined(IR_SEND_PIN)
3
6 #define IR_SEND_PIN          3
3
3 #endif
7
3
3 /*
8
3 * Specify DistanceWidthProtocol for decoding. This must be done before the #include
3 <IRremote.hpp>
9
```

```
4 */
0
4 #define DECODE_DISTANCE_WIDTH // Universal decoder for pulse distance width
4 protocols
1
4 //
4
2 #if !defined(RAW_BUFFER_LENGTH)
4 #   if RAMEND <= 0x4FF || RAMSIZE < 0x4FF
3 #define RAW_BUFFER_LENGTH 120
4 #   elif RAMEND <= 0xAFF || RAMSIZE < 0xAFF // 0xAFF for LEONARDO
4 #define RAW_BUFFER_LENGTH 400 // 600 is too much here, because we have additional
4 uint8_t rawCode[RAW_BUFFER_LENGTH];
5
4 #   else
6 #define RAW_BUFFER_LENGTH 750
4 #   endif
7
4 #endif
8
4 // #define NO_LED_FEEDBACK_CODE // saves 92 bytes program memory
9 // #define RECORD_GAP_MICROS 12000 // Default is 5000. Activate it for some LG air
5 conditioner protocols
0 // #define SEND_PWM_BY_TIMER // Disable carrier PWM generation in software and
5 use (restricted) hardware PWM.
1 // #define USE_NO_SEND_PWM // Use no carrier PWM, just simulate an active low
5 receiver signal. Overrides SEND_PWM_BY_TIMER definition
2
5 // #define DEBUG // Activate this for lots of lovely debug output from the decoders.
3
5
4 #include <IRremote.hpp>
5
5 #define SEND_BUTTON_PIN APPLICATION_PIN
6
5 #define DELAY_BETWEEN_REPEATS_MILLIS 70
7
```

```
5
8
// Storage for the recorded code, pre-filled with NEC data
5
9 IRRawDataType sDecodedRawDataArray[RAW_DATA_ARRAY_SIZE] = { 0x7B34ED12 }; // address
0x12 command 0x34
6
0 DistanceWidthTimingInfoStruct sDistanceWidthTimingInfo = { 9000, 4500, 560, 1690,
560, 560 }; // NEC timing
6
1 uint8_t sNumberOfBits = 32;
6
2 bool sSendButtonWasActive;
6
3
void setup() {
6
4     pinMode(SEND_BUTTON_PIN, INPUT_PULLUP);
6
5
    Serial.begin(115200);
6
6 #if defined(__AVR_ATmega32U4__) || defined(SERIAL_PORT_USBVIRTUAL) ||
defined(SERIAL_USB) /*stm32duino*/ || defined(USBCON) /*STM32_stm32*/ ||
6 defined(SERIALUSB_PID) || defined(ARDUINO_attiny3217)
7
    delay(4000); // To be able to connect Serial monitor after reset or power up and
6 before first print out. Do not wait for an attached Serial Monitor!
8
#endif
6
9     // Just to know which program is running on my Arduino
7
    Serial.println(F("START " __FILE__ " from " __DATE__ "\r\nUsing library version
0 " VERSION_IRREMOTE));
7
1
    // Start the receiver and if not 3. parameter specified, take LED_BUILTIN pin
7 from the internal boards definition as default feedback LED
2
    IrReceiver.begin(IR_RECEIVE_PIN, ENABLE_LED_FEEDBACK);
7
    Serial.println(F("Ready to receive pulse distance/width coded IR signals at pin
3 " STR(IR_RECEIVE_PIN)));
7
4
    IrSender.begin(); // Start with IR_SEND_PIN as send pin and enable feedback LED
7 at default feedback LED pin
5
```

```
7   Serial.print(F("Ready to send IR signals at pin " STR(IR_SEND_PIN) " on press of
6button at pin "));

7   Serial.println(SEND_BUTTON_PIN);
7   }
7
8
7 void loop() {
9
8   // If button pressed, send the code.
0
8   bool tSendButtonIsActive = (digitalRead(SEND_BUTTON_PIN) == LOW); // Button pin
8 is active LOW
1
8
2   /*
8   * Check for current button state
3
8   */
4   if (tSendButtonIsActive) {
8       if (!sSendButtonWasActive) {
5           Serial.println(F("Stop receiving"));
8
6           IrReceiver.stop();
8
8       }
7       /*
8       * Button pressed -> send stored data
8
8       */
9       Serial.print(F("Button pressed, now sending "));
9       Serial.print(sNumberOfBits);
0       Serial.print(F(" bits 0x"));
9       Serial.print(sDecodedRawDataArray[0], HEX);
1       Serial.print(F(" with sendPulseDistanceWidthFromArray timing="));
9
2       IrReceiver.printDistanceWidthTimingInfo(&Serial, &sDistanceWidthTimingInfo);
9
3       Serial.println();
3
```



```
9     Serial.flush(); // To avoid disturbing the software PWM generation by serial
4 output interrupts
9
5     IrSender.sendPulseDistanceWidthFromArray(38, &sDistanceWidthTimingInfo,
9 &sDecodedRawDataArray[0], sNumberOfBits,
6
9 #if defined(USE_MSB_DECODING_FOR_DISTANCE_DECODER)
7     PROTOCOL_IS_MSB_FIRST
9
9 #else
8     PROTOCOL_IS_LSB_FIRST
9
9 #endif
9
1     , 100, 0);
0
0     delay(DELAY_BETWEEN_REPEATS_MILLIS); // Wait a bit between retransmissions
1
0
1 } else if (sSendButtonWasActive) {
1
0     /*
2     * Button is just released -> activate receiving
1
0     */
3     // Restart receiver
1
0     Serial.println(F("Button released -> start receiving"));
4
0     IrReceiver.start();
1
0
5 } else if (IrReceiver.decode()) {
1
0     /*
6     * Button is not pressed and data available -> store received data and resume
1 the same IR remote / protocol
0
0     */
7
0     IrReceiver.printIRResultShort(&Serial);
1
0     if (IrReceiver.decodedIRData.protocol != UNKNOWN) {
```

```
1      IrReceiver.printIRSendUsage(&Serial);
0
8
1      if (memcmp(&sDistanceWidthTimingInfo,
0 &IrReceiver.decodedIRData.DistanceWidthTimingInfo,
9      sizeof(sDistanceWidthTimingInfo)) != 0) {
1          Serial.print(F("Store new timing info data="));
1
0      IrReceiver.printDistanceWidthTimingInfo(&Serial,
&IrReceiver.decodedIRData.DistanceWidthTimingInfo);
1
1      Serial.println();
1
1      sDistanceWidthTimingInfo =
1 IrReceiver.decodedIRData.DistanceWidthTimingInfo; // copy content here
1
1      } else {
2
1          Serial.print(F("Timing did not change, so we can reuse already
1 stored timing info.));
3      }
1
1      if (sNumberOfBits != IrReceiver.decodedIRData.numberOfBits) {
4
1          Serial.print(F("Store new numberOfBits="));
1
1          sNumberOfBits = IrReceiver.decodedIRData.numberOfBits;
1
5          Serial.println(IrReceiver.decodedIRData.numberOfBits);
1
1      }
1
1      if (sDecodedRawDataArray[0] !=
6 IrReceiver.decodedIRData.decodedRawDataArray[0]) {
1
1          *sDecodedRawDataArray =
1 *IrReceiver.decodedIRData.decodedRawDataArray; // copy content here
7
1          Serial.print(F("Store new sDecodedRawDataArray[0]=0x"));
1
1          Serial.println(IrReceiver.decodedIRData.decodedRawDataArray[0],
8 HEX);
1
1      }
1
9      }
1
1      IrReceiver.resume(); // resume receiver
2
0      }
```

```
1  
2  
1   sSendButtonWasActive = tSendButtonIsActive;  
1   delay(100);  
2 }  
2
```

The sketch is similar to the previous one in some respects, with one important consideration. Line 56 has a constant *DECODE_DISTANCE_WIDTH*. This puts the IR receiver into distance width protocol mode, which allows it to capture the distance between raw data pulses. It is important to put this line before the call to the IRremote library.

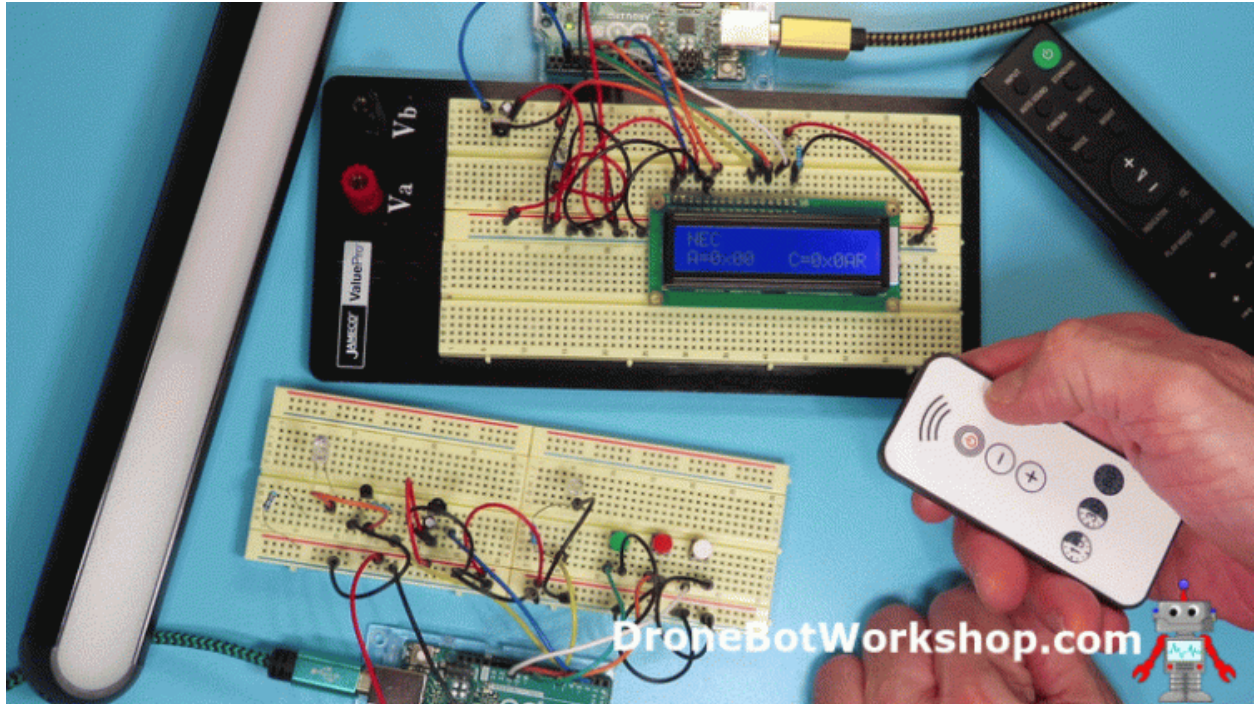
Once again, you need to change the button pin constant *SEND_BUTTON_PIN*. Change it from *APPLICATION_PIN* to 10.

The key to understanding this sketch is on lines 81-84. The three variables here are what we store every time we receive a valid IR packet, and it is what we transmit when the button is pressed. We store an array of the data, a data structure with timing info, and an integer with the number of bits.

In the Setup, we do the usual serial monitor setup, as well as setting the pin mode of the pushbutton to an input with pull-up. We also start the IR receiver and sender.

In the Loop, we check to see if the button was pressed. If it was, we send the packet using the *IrSender.sendPulseDistanceWidthFromArray* function.

Otherwise, we are in receive and store mode. We look for a valid packet, and when we find one, we extract its timing and data information and calculate the number of bits. Then, we store those values to be used for the next send.



Load the sketch to the Arduino and test it out. If you observe the serial monitor, you'll see that every IR signal is treated as raw data.

I have some projects coming up in a future article and video that use this technique. It's a great way to handle those remotes you can't decode otherwise.

ESP32 Remote Control

We'll finish everything up with a small project, a remote control based on the ESP32.

This remote will actually mimic the functionality of the 3-button remote we constructed with the Arduino, except we won't have any physical buttons. Instead, we will use the WiFi capabilities of the ESP32 to have a web-based interface. This will allow you to expand upon the design to create a custom remote with as many buttons as you desire.

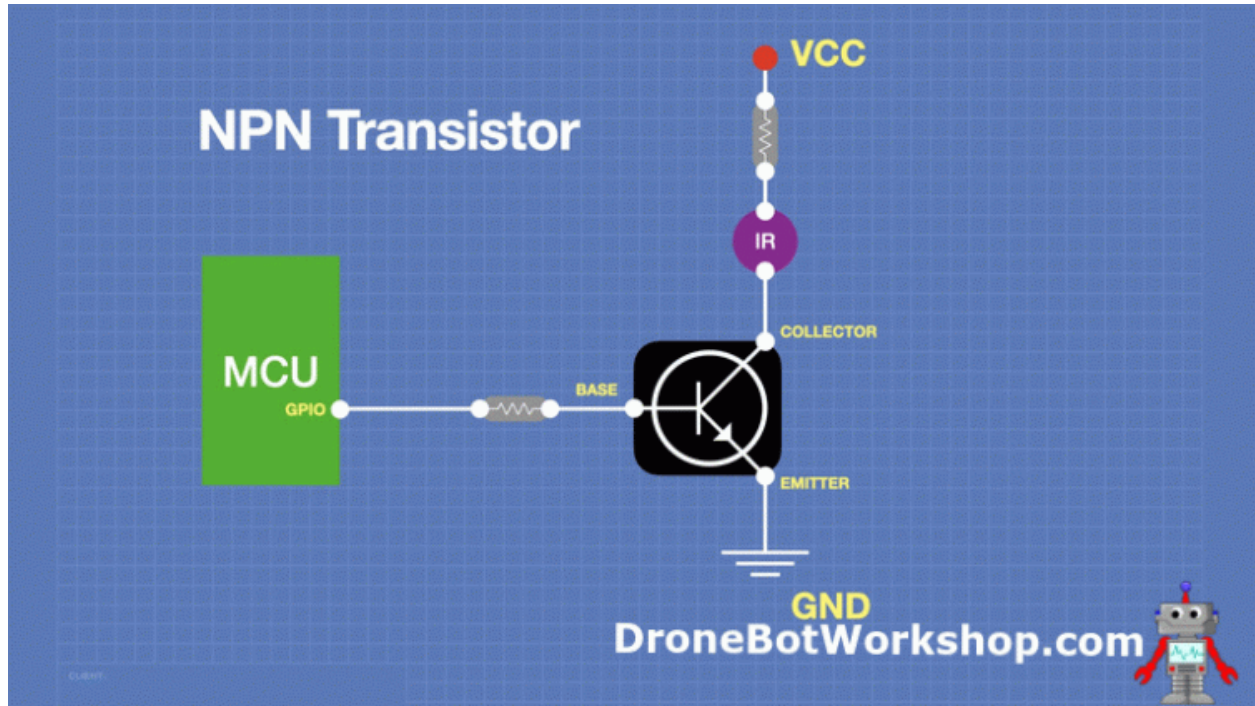
Our ESP32 will be set up as a WiFi access point, so this design doesn't need to connect to a WiFi network.

One feature we will implement in this design that was missing from the previous one is the repeat function when a button is held down. As we can't implement it exactly, we will be "faking" it!

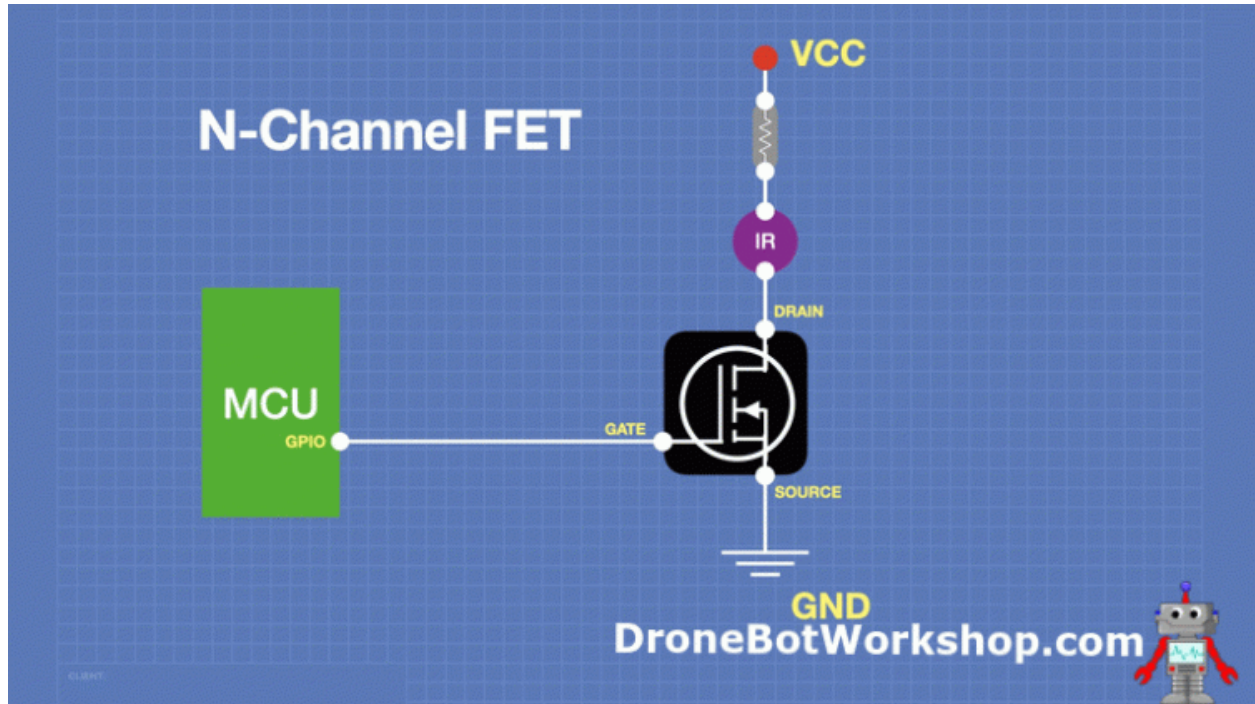
ESP32 Remote Control Hookup

The hookup for the ESP32 remote control is very simple. All we really need is an IR emitter LED and a method of driving it.

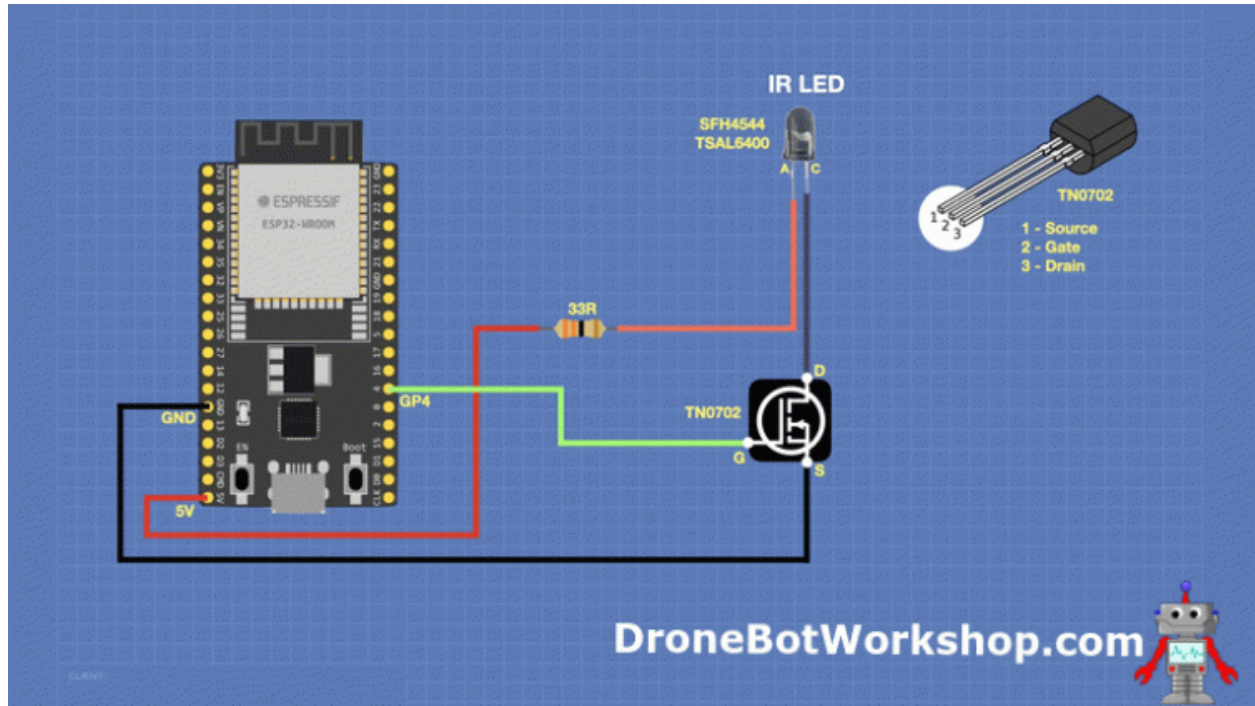
We can just drive the IR LED with the 2N2222 we used earlier; that will work fine. You may wish to drop the 1k resistor on the base to 680 ohms, but otherwise, the hookup is the same. We still use 5-volts for the LED, even though the ESP32 is a 3.3-volt microcontroller.



Another method of driving the infrared emitter LED is to use a FET (Field Effect Transistor). This has efficiency advantages and also eliminates the need for a resistor on the GPIO pin. This connection is illustrated here:



Here is how we will hook up our ESP32. Any ESP32 module will work for this design. The TN0702 FET specified in the diagram is capable of 20 volts at about half an ampere, so it can be used with multiple emitter LEDs if you wish to experiment.



ESP32 Remote Control Code

Here is the code we will use for the ESP32 remote control. You can use it as the basis for designing your own remote.

For more projects and tutorials visit the DroneBot Workshop - <https://dronebotworkshop.com>

```
1 /*
2   ESP32 IR Remote
3   ir-remote-esp32.ino
4   IR Remote control with web interface
5   Uses Arduino-IRremote Library -
6   https://github.com/Arduino-IRremote/Arduino-IRremote
7   Uses ESPAsyncWebServer Library - https://github.com/me-no-dev/ESPAsyncWebServer
8   Uses AsyncTCP Library - https://github.com/me-no-dev/AsyncTCP
9
10  DroneBot Workshop 2023
11  https://dronebotworkshop.com
12 */
13
14 // Include required libraries
15 #include <Arduino.h>
16 #include <WiFi.h>
17 #include <AsyncTCP.h>
18 #include <ESPAsyncWebServer.h>
19 #include "TinyIRSender.hpp"
20
21 // WiFi Access Point credentials (use NULL for password if none required)
22 const char *ssid = "ESP32-REMOTE"; // Enter SSID here
23 const char *password = "12345678"; //Enter Password here
24
25 // Async web server at port 80
26 AsyncWebServer server(80);
27
28 // Define IR LED pin on GPIO4
29 #define IR_SEND_PIN 4
```

<https://dronebotworkshop.com>

```
2
2
// Button code variables (change to match your remote control)
2
3 uint16_t codePower = 0x0A;
2
uint16_t codeMinus = 0x09;
4
uint16_t codePlus = 0x0B;
2
5
// Variables to hold complete send parameters (initialized with NEC 00 00)
2
6 uint16_t sAddress = 0x00;
2
volatile uint16_t sCommand = 0x00;
7
uint16_t sRepeats = 0;
2
8
// Flag for button held down
2
9 volatile bool buttonDown = false;
3
0
// HTML web page
3
1 const char index_html[] PROGMEM = R"rawliteral(
3
3 <!DOCTYPE HTML><html>
2
    <head>
3
    <title>Remote Control</title>
3
    <meta name="viewport" content="width=device-width, initial-scale=1">
4
    <style>
3
        body { font-family: Times New Roman; text-align: center; margin:0px auto;
5 padding-top: 30px;}
3
        .button {
6
            padding: 10px 20px;
3
            font-size: 24px;
7
            text-align: center;
3
            outline: none;
8
            color: #fff;
3
9
```

```
4     background-color: #ff0522;
0
4     border: none;
1
4     border-radius: 5px;
2
4     cursor: pointer;
2
4     -webkit-touch-callout: none;
4
4     -webkit-user-select: none;
3
4     -khtml-user-select: none;
4
4     -moz-user-select: none;
4
4     -ms-user-select: none;
5
4     user-select: none;
6
4     -webkit-tap-highlight-color: rgba(0,0,0,0);
}
7
4     .button:hover {background-color: #ff0522 }
8
4     .button:active {
8
4         background-color: #1fe036;
9
4         transform: translateY(2px);
5
4     }
0
4     </style>
5
1     </head>
5
4     <body>
2
4     <h1>Lamp Control</h1>
5
4     <p><button class="button" onmousedown="toggleCheckbox('pON');"
3 ontouchstart="toggleCheckbox('pON');" onmouseup="toggleCheckbox('pOFF');"
5 ontouchend="toggleCheckbox('pOFF');">Power</button></p>
4
4     <p><button class="button" onmousedown="toggleCheckbox('uON');"
5 ontouchstart="toggleCheckbox('uON');" onmouseup="toggleCheckbox('uOFF');"
5 ontouchend="toggleCheckbox('uOFF');">Up</button></p>
5
4     <p><button class="button" onmousedown="toggleCheckbox('dON');"
6 ontouchstart="toggleCheckbox('dON');" onmouseup="toggleCheckbox('dOFF');"
5 ontouchend="toggleCheckbox('dOFF');">Down</button></p>
5
7
```

```
5  <script>
8
    function toggleCheckbox(x) {
5
        var xhr = new XMLHttpRequest();
9
        xhr.open("GET", "/" + x, true);
6
        xhr.send();
0
    }
6
    </script>
1
2  </body>
6</html>rawliteral";
3
6
4 void setup() {
6
5    // Start serial monitor
6
    Serial.begin(115200);
6
6
7    WiFi.mode(WIFI_AP);
6
    WiFi.softAP(ssid, password);
8
6
9    Serial.println();
7
    IPAddress IP = WiFi.softAPIP();
0
    Serial.print("Access Point IP address: ");
7
    Serial.println(IP);
1
7
2    // Send web page to client
7
    server.on("/", HTTP_GET, [] (AsyncWebServerRequest *request) {
3
        request->send_P(200, "text/html", index_html);
7
    });
4
7
5
```

```
7 // Receive an HTTP GET request from power button pressed
6
server.on("/pON", HTTP_GET, [] (AsyncWebServerRequest *request) {
7
    Serial.println("PWR ON");
7
    sCommand = codePower;
8
    buttonDown = true;
7
    request->send(200, "text/plain", "OK");
9
});
8
0
8 // Receive an HTTP GET request from power button released
1
server.on("/pOFF", HTTP_GET, [] (AsyncWebServerRequest *request) {
8
    Serial.println("PWR OFF");
2
    sCommand = 0x00;
8
    buttonDown = false;
3
    request->send(200, "text/plain", "OK");
8
    });
4
8
5
8 // Receive an HTTP GET request from up button pressed
6
server.on("/uON", HTTP_GET, [] (AsyncWebServerRequest *request) {
8
    Serial.println("UP ON");
7
    sCommand = codePlus;
8
    buttonDown = true;
8
    request->send(200, "text/plain", "OK");
9
    });
9
0
9 // Receive an HTTP GET request from up button released
1
server.on("/uOFF", HTTP_GET, [] (AsyncWebServerRequest *request) {
9
    Serial.println("UP OFF");
2
    sCommand = 0x00;
9
3
```

```
9   buttonDown = false;
4
   request->send(200, "text/plain", "OK");
9
5   });

9
6   // Receive an HTTP GET request from down button pressed
9
   server.on("/dON", HTTP_GET, [] (AsyncWebServerRequest *request) {
7
       Serial.println("DOWN ON");
9
       sCommand = codeMinus;
8
       buttonDown = true;
9
       request->send(200, "text/plain", "OK");
9
1      });
0
0

1      // Receive an HTTP GET request from down button released
0
1      server.on("/dOFF", HTTP_GET, [] (AsyncWebServerRequest *request) {
1
           Serial.println("DOWN OFF");
0
           sCommand = 0x00;
2
           buttonDown = false;
1
           request->send(200, "text/plain", "OK");
0
3          });
1
0
4
           // Start the server
1
           server.begin();
0
5      }

1
0
6 void loop() {
1
0
7     // Ignore if command value is zero

    if (sCommand != 0x00) {
```

```
1
0
8 // See if a button was pressed
1 if (buttonDown) {
0
9
1 // Send IR code
1 sendNEC(IR_SEND_PIN, sAddress, sCommand, sRepeats);
0
1
1 // Print send code to serial monitor
1 Serial.print("Sent command ");
1 Serial.println(sCommand, HEX);
2
1 // Short delay for repeat
1 delay(110);
3
1
1 // See if button is still held down
4 while (buttonDown && sCommand != 0x00) {
1 // Same value, Send IR code with repeats
5 sendNEC(IR_SEND_PIN, sAddress, sCommand, 1);
1 Serial.print("Sent repeat command ");
1 Serial.println(sCommand, HEX);
6 delay(240);
1
7 }
1
1 }
8
1 }
1 }
9 }
```

For more projects and tutorials visit the DroneBot Workshop - <https://dronebotworkshop.com>

Before you can use this code, you will need to install a couple of libraries, ones that are not in the Arduino IDE Library Manager. You can get both of them on GitHub

- [ESPAsyncWebServer Library](#)
- [AsyncTCP Library](#)

Both of these libraries are required to allow asynchronous communications between the web client and the server. This is necessary, as we don't want to have to refresh our web page every time we press a button.

We'll include all of the required libraries and then set up the credentials for the access point that the ESP32 will create. You can change these to any value you want; the password should be at least eight characters. You can also use a null value if you don't want a password.

Next, we start the asynchronous web server on port 80.

The next bit of the code might look familiar, as it's essentially the same variables we used in the sketch with the tree pushbuttons. These define our IR emitter pin, the codes we are using, and some variables to hold the transmitted data values. There is also a flag to indicate that a button is being held down.

The next constant is actually the entire HTML code for the web page, including CSS and JavaScript. It is easy to read and modify.

The JavaScript is the key to making things work. Each button looks for events caused by buttons being either clicked or pressed (you need both for web pages and phones). Those events call a JavaScript function (*toggleCheckbox*) that sends data to the server using an asynchronous method.

For more projects and tutorials visit the DroneBot Workshop - <https://dronebotworkshop.com>

In Setup, we start the serial monitor and then set the ESP32 WiFi up as an Access Point. Once the access point is established, we print our server's IP address to the serial monitor. Our client will need this to connect.

We then define how the server should respond to specific asynchronous transactions. As most of the transactions are for buttons, we will use them to determine which button was pressed and if it is being held down. We set the *buttonDown* boolean and command value according to the button and its state.

After defining the responses, we start the server.

In the Loop, we test to see if a button was pressed. If it was, we send the values for the address and command using *sendNEC* (check the documentation if you use another protocol).

We also delay and see if the button is still held down. If it is, we need to send a repeat code.

As the *TinyIRSender* library does not have a function to only send a repeat code, we actually send a command again, followed by a repeat code. So, it is not a “true” repeat, but it accomplishes the same thing!

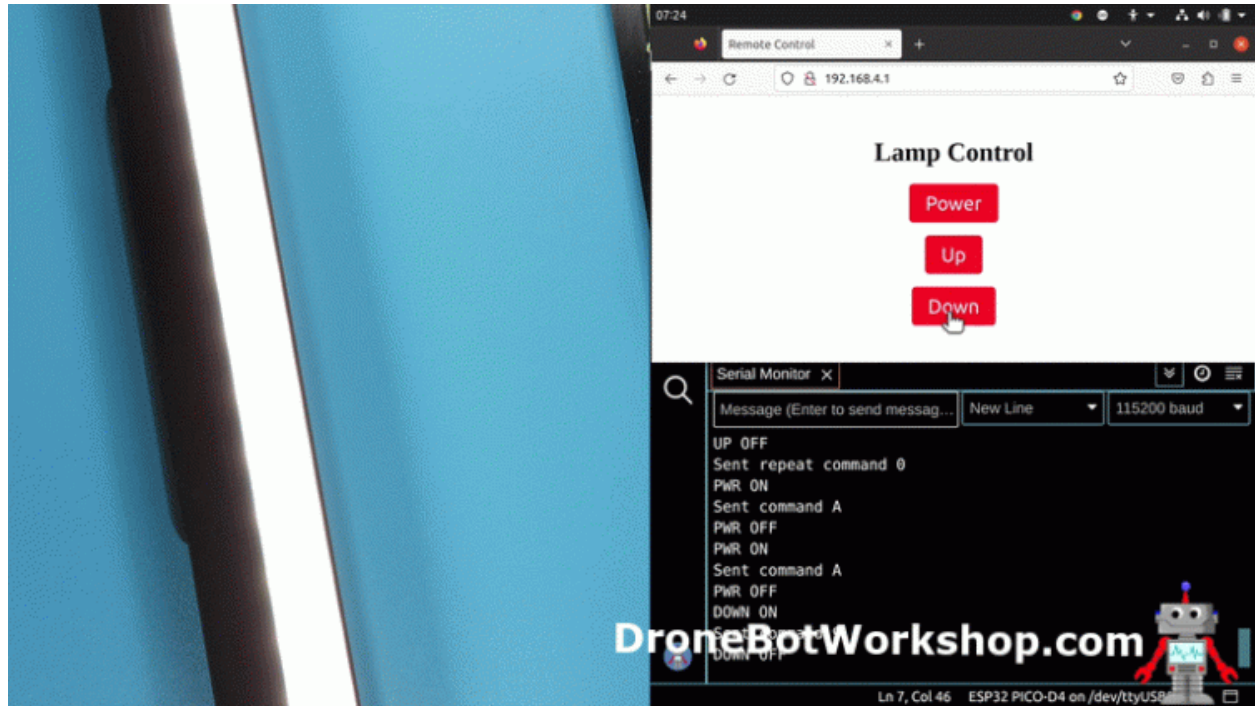
Load the code and give it a try. You will have to connect to the WiFi access point first with a phone, tablet, or computer. Afterward, go to the address in the serial monitor using a web browser.

The page does not have a security certificate, so if your browser gives you grief, try another one. Firefox accepts pages without certificates.

Once you are there, the page should display.

<https://dronebotworkshop.com>

For more projects and tutorials visit the DroneBot Workshop - <https://dronebotworkshop.com>



Try it out; it should duplicate the function of your remote control

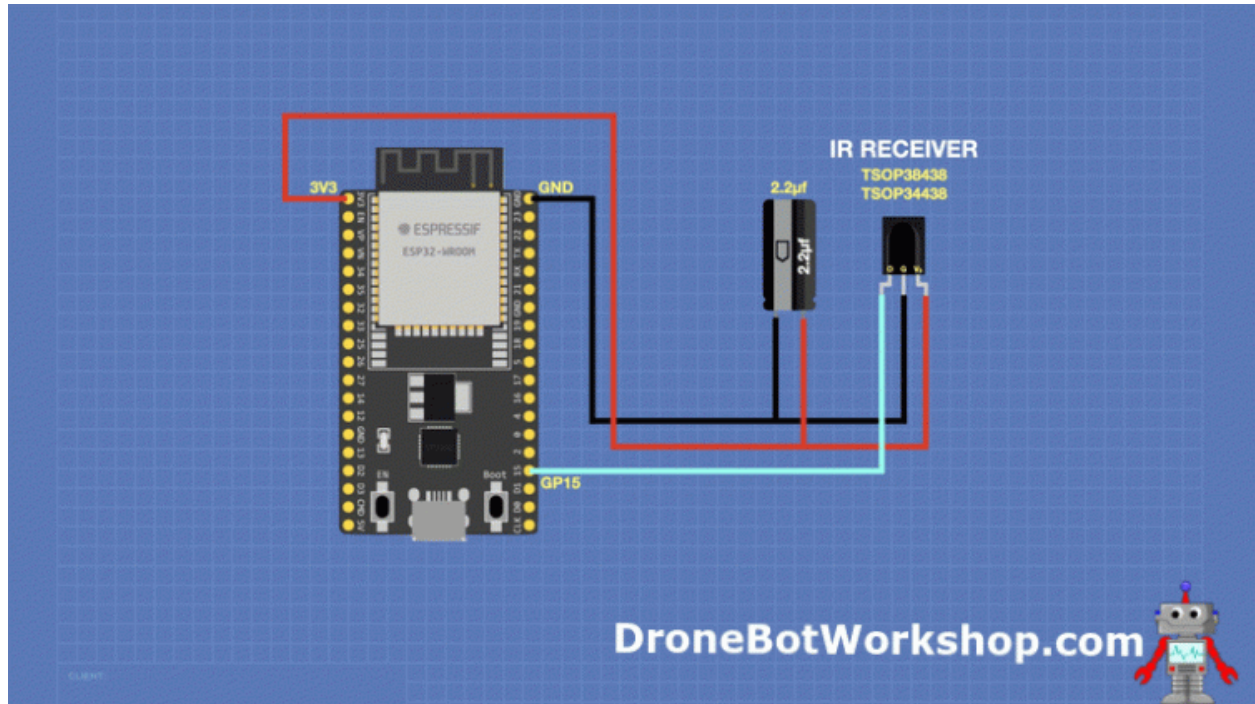
Of course, you can extend this technique to have more buttons and functions.

ESP32 Receiver Hookup

One final hookup I will leave you with is for the ESP32 and an IR receiver.

Unlike the Arduino, which operates at 5 volts, the ESP32 will require a more modern IR Sensor that can work with lower voltages. The TSOP38438 we used earlier will work great, as will most 3.3-volt IR sensor modules.

<https://dronebotworkshop.com>



As with the previous hookups, we are placing a 2.2uf capacitor near the IR sensor to reduce any power supply noise.

With this hookup, you can run the example code included with both libraries.

Conclusion

Interfacing a microcontroller with infrared remote controls and IR-controlled devices opens up dozens of project possibilities. You could, for example, have some lights that are controlled by your television remote, coming on when the TV is switched on.

I have three more IR remote projects that I started working on while I was researching this article; you'll be seeing them soon. And I'm sure you have several ideas for adapting this technology to your own needs.

For more projects and tutorials visit the DroneBot Workshop - <https://dronebotworkshop.com>

So get out there and use invisible light beans to control your world. All you need is a microcontroller and a few IR devices, and you're set to take command.

Just remember to get off the couch once in a while!

Parts List

Here are some components you might need to complete the experiments in this article.

Vishay TSOP34438 [DigiKey](#)

Vishay TSOP38238 [DigiKey](#)

OSRAM SFH 4544 [DigiKey](#)

Resources

Code Samples – All of the code used in the article in one easy-to-use ZIP file!

Article PDF – A PDF version of this article in a ZIP file.

[Arduino IRremote Library](#) – Arduino IRremote Library on GitHub

[IRMP Library](#) – Infrared Multi-Protocol Decoder + Encoder Library on GitHub

[ESPAsyncWebServer Library](#) – ESPAsyncWebServer Library on GitHub

[AsyncTCP Library](#) – AsyncTCP Library on GitHub

[Vishay TSOP34438](#) – IR Receiver Module Specs

[Vishay TSOP38238](#) – IR Receiver Module Specs

[TL1838](#) – IR Receiver Module Specs

[OSRAM SFH 4544](#) – 940nm IR Emitter LED